Timed coloured Petri nets and their application to logistics

The study reported in this monograph is sponsored by the TNO Institute for Production and Logistics (IPL) as part of the TASTE project.

TIMED COLOURED PETRI NETS AND THEIR APPLICATION TO LOGISTICS

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
donderdag 17 september 1992 om 16.00 uur

door

Willibrordus Martinus Pancratius van der Aalst

geboren te Eersel

Dit proefschrift is goedgekeurd
door de promotoren
prof. dr. J. Wessels
en
prof. dr. K.M. van Hee

Contents

Chapter 1

Introduction

1.1 Problem statement

Recently, logistics has become an important issue in many organizations. This is a direct consequence of the fact that modern organizations are required to offer a wide variety of products, in less time and at reduced prices. To improve their logistics function, many organizations have integrated the control of the logistic activities such as production, transportation, storage, acquisition and distribution. This integration complicates the management of the logistic processes. The complexity of the control problems encountered in logistics urges the necessity of an integrated framework for the modelling and analysis of logistic systems.

This monograph focuses on the modelling and analysis of complex logistic systems and outlines solutions based on a *timed coloured Petri net* model. Although these solutions are useful in the context of logistics, their application is not limited to the logistic domain. Examples of other application domains which may benefit from the results presented in this monograph are: flexible manufacturing systems, distributed information systems and real-time systems. In fact most of the results apply to systems which are:

dynamic The systems we are interested in are subject to changes. At any moment the system has a certain state, at a later time this state may have changed.

discrete We restrict ourselves to discrete systems, i.e. changes in the system occur discontinuously. These changes only happen at a finite number of time points.

distributed A distributed system is composed of a number of autonomous subsystems which interact and share resources in performing a specific task. These subsystems are often physically distributed.

In other words: we consider distributed systems that change in a discrete fashion. We call these systems *discrete dynamic systems*.

We use a Petri net based approach to the modelling and analysis of these discrete dynamic systems. Petri nets are appropriate for the modelling of distributed systems, since they allow for the representation of parallelism and synchronization. However, the classic Petri net model is unsuitable for the modelling of systems having large state spaces or a complex temporal behaviour. Therefore, we have developed a Petri net model extended with *time* and *colour*. This model is the foundation of a framework that has been developed to solve problems related to the design and control of complex discrete dynamic systems.

In this monograph, we focus on two important aspects of this framework:

modelling There are several reasons for modelling a system, e.g. to create and evaluate a design of a new system, to compare alternative designs and to investigate possible improvements in a real system. Model building forces us to organize, evaluate and examine the validity of our thoughts. This way modelling reveals errors and possible improvements.

The outcome of any modelling process is a 'model'. We distinguish three kinds of models: (1) informal models, (2) mathematical models and (3) formal specifications.

An informal model is a verbal and/or graphical description of the system under consideration. Such a model lacks formal semantics.

Mathematical models are those in which one or more aspects of a system are represented by mathematical entities, like: equations, matrices, relations, Markov chains, graphs, etc. These models are often an abstraction of the real system in which simplifying assumptions are required if the model is to be solvable.

A formal specification is a precise and structured description of (aspects of) a system. Such a specification is an abstraction of the real system, expressed in a specification language having a predefined syntax and semantics. Unlike most mathematical models, a formal specification cannot be 'solved' analytically. However, most formal specifications are based on a mathematical model allowing for one or more kinds of analysis. Although analysis is possible by analysing the underlying model, the primary function of a formal specification is to provide a concise and unambiguous description of the system (i.e. a 'blueprint').

In this monograph we focus on specifications based on timed coloured Petri nets. A timed coloured Petri net is a mathematical model which is suitable for the modelling of discrete dynamic systems.

The development of a good specification of a complex discrete dynamic system is often time consuming and requires considerable knowledge and experience. Therefore, there is a need for concepts and tools to facilitate the modelling process. Since we concentrate on logistics, we are particularly interested in

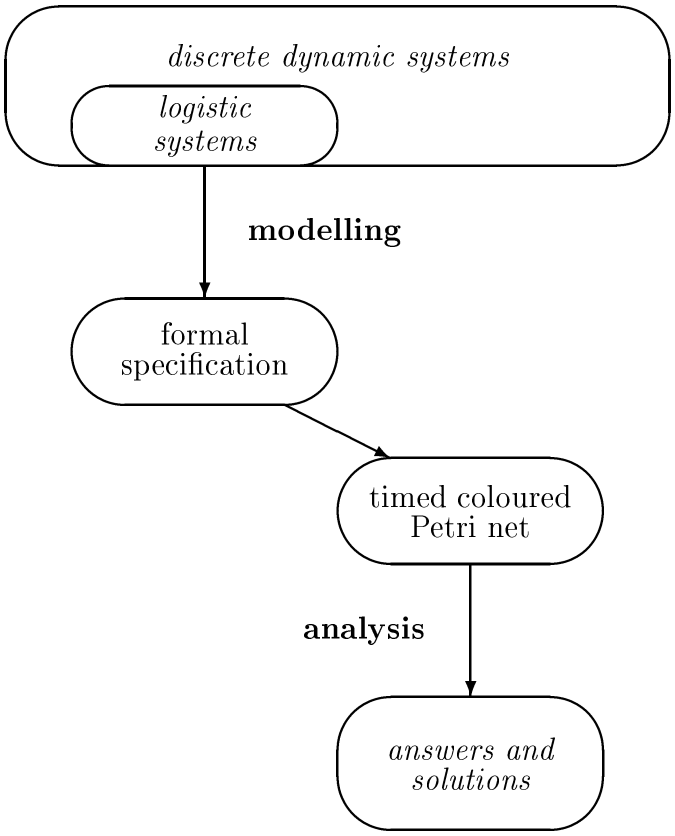


Figure 1.1: A survey of this monograph

concepts useful for the modelling of complex logistic systems. Consequently, some of the concepts we have developed apply to logistic systems in particular.

analysis The outcome of the modelling process is a specification which corresponds to a timed coloured Petri net. Analysis of this net may be useful to verify its correctness and to make statements about the performance of the system. It also helps the modeller to understand the behaviour of the system.

To analyse the dynamic behaviour of a timed coloured Petri net, we need analysis methods. Simulation is a suitable technique for the analysis of this type of nets. Although simulation is flexible and easy to use, there is an urge for other techniques which exploit the features of Petri nets extended with ‘time’ and ‘colour’. Many analysis techniques developed for classic Petri nets have been extended for coloured nets. However, these techniques cannot be used to analyse the *temporal* behaviour of a timed coloured Petri net.

Therefore, we have developed a number of powerful analysis methods, three of which are presented in this monograph.

The purpose of this monograph is summarized in figure 1.1. On the one hand this monograph discusses concepts and tools to facilitate the modelling of logistic systems, on the other hand it provides methods to analyse timed coloured Petri nets. These results are outlined in this monograph and are based on concepts from Petri net theory, systems analysis and knowledge of logistics as an application domain.

1.2 Petri nets

The systems we consider are often very complex, large, discrete dynamic systems of many interacting components. The components of such a system exhibit *concurrency* or *parallelism*, i.e. activities of one component may occur simultaneously with other components. The components of the system interact and sometimes they have to *synchronize*, i.e. one component waits for the other in order to execute an activity simultaneously. The Petri net formalism (Petri [102], Reisig [111]) was one of the first approaches introduced for dealing with concurrency and synchronization.

Historically speaking, Petri nets originate from the early work of Carl Adam Petri ([101]). Petri's work came to the attention of Holt and others of the Information System Theory Project of Applied Data Research, Inc, in the United States. Much of the early theory and notation has been developed by this group ([65]). The work of Petri also came to the attention of Project MAC at the Massachusetts Institute of Technology (MIT), resulting in a number of publications and reports. Since the late-1970's, the use and study of Petri nets has increased considerably. Especially Europeans have been very active in the field of Petri nets. Research on and the application of Petri nets have become widespread activities. A review of the history of Petri nets and an extensive bibliography is given by Murata in [93].

The classic (or basic) Petri net is a directed bipartite graph with two node types called *places* and *transitions*. The nodes are connected via directed *arcs*. Connections between two nodes of the same type are not allowed. Places are represented by circles and transitions by bars. Places may contain zero or more *tokens*, drawn as black dots. The number of tokens may change during the execution of the net. A place p is called an *input place* of a transition t if there exists a directed arc from p to t. A place p is called an *output place* of a transition t if there exists a directed arc from t to p. Each transition puts a *weight* to each of its input and output places, i.e. each arc is labelled with a weight (positive integer). A transition is called *enabled* if each of its input places contains at least a number of tokens equal to its weight. In other words, a transition is enabled if all input places contain (at least) the specified number of tokens. An enabled transition can *fire*. Firing a transition t means consuming tokens from the input places and producing tokens for the output places, i.e. t 'occurs'. The number of tokens produced for each of the output places is equal to the weight of the corresponding arc. A state of a Petri net is a distribution of tokens over the places. Many authors use the term *marking* to denote the state of a basic Petri net. A *firing sequence* is a sequence of states $s_1, s_2, s_3, ..,$ such that any state s_i is followed by a state s_{i+1}, resulting from the firing of some enabled transition in state s_i.

For a Petri net which models a discrete dynamic system, we are often interested in properties, such as boundedness, liveness, safety and freedom of deadlock. Moreover, given an initial state (marking), we are also interested in the *reachability set*, i.e. the set of all states visited by some firing sequence starting in this initial state. To answer these questions, several analysis techniques have been suggested.

Most of the analysis techniques described in literature, either generate a reachability graph or involve linear algebraic techniques.
A *reachability graph* is a graph representation of the reachable states and can be used to answer a variety of questions. Several reduction techniques have been developed to reduce the size of such a graph.
Linear algebraic techniques are often used to calculate *invariants*. Note that a marking can be represented as a vector, and a Petri net can be represented as a set of linear algebraic equations. Invariants are characteristic solutions of these equations. Therefore, it is possible to compute them by linear algebraic techniques. An example of an invariant is a so-called place invariant, this invariant assigns a weight to each place, such that the weighted token count remains constant during the execution of any firing sequence.

For an introduction to basic Petri nets and their analysis we refer to Reisig [111] and Peterson [100].

Since the beginning of the 1970's the study of Petri nets has developed in two directions: *pure* Petri net theory and *applied* Petri net theory.
The first line of research concentrates on the fundamental theory of Petri nets. People working on this line of research are mainly concerned with the development of a firm mathematical foundation of Petri net theory. Although the results of this kind of research are useful, many techniques and concepts developed in this area are difficult to apply to problems encountered in practice.
The second line of research is concerned with the application of Petri nets to the modelling and analysis of systems. Typical application areas are communication protocols, computer systems, distributed systems, production systems and flexible manufacturing systems. In applying Petri nets, it is often necessary to extend the basic Petri net model.[1] These extensions do not allow the use of many techniques developed in the field of pure Petri net theory. Fortunately, many of these techniques have been generalized to coloured Petri nets.

Both directions did not lead to a comprehensive framework of Petri nets, which fully utilizes the analysis capabilities and is applicable in practice. Consequently, there is still a great gulf between pure and applied Petri net theory. This monograph describes concepts and techniques which are useful for bridging this gulf.

[1]Note that we use the term 'Petri net *model*' to denote a formal definition of Petri nets, such a model is in fact a meta-model, since it is *used* to describe models of systems.

1.3 Time and colour in Petri nets

The basic Petri net model is not suitable for the modelling of many systems encountered in logistics, production, communication, flexible manufacturing and information processing. Petri nets describing real systems tend to be complex and extremely large. Sometimes, it is even impossible to model the behaviour of the system accurately. To solve these problems many authors propose extensions of the basic Petri net model.

We distinguish two kinds of extensions: (1) extensions to increase the modelling power and (2) extensions to merely facilitate the user in making more succinct and manageable models. Examples of extensions that do not increase the power of a Petri net model are multiple arcs and places with capacity constraints (see Murata [93]). On the other hand there are extensions, such as inhibitor arcs ('zero test') and priorities (Peterson [100], Pagnoni [97]), that do increase the modelling power. When adding these extensions, careful attention must be paid to the tradeoff between modelling and analysis capability. That is, the more general the model, the more difficult it is to analyse.

The approach presented in this monograph is based on a *timed coloured* Petri net model, called the *Interval Timed Coloured Petri Net* (ITCPN) model. We start with an informal introduction to the ITCPN model by relating it to other timed and/or coloured Petri net models known in literature.

1.3.1 Adding colour

Many authors have extended the basic Petri net model with *coloured* or *typed tokens* ([132], [99], [46], [70], [71], [53]). In these models tokens have a value, often referred to as 'colour'. There are several reasons for such an extension. One of these reasons is the fact that (uncoloured) Petri nets tend to become too large to handle. Another reason is the fact that tokens often represent objects or resources in the modelled system. As such, these objects may have attributes, which are not easily represented by a simple Petri net token.
These 'coloured' Petri nets allow the modeller to make much more succinct and manageable descriptions, therefore they are called 'high-level' nets. Although Zervos ([132]) presented a coloured Petri net in 1977, the first well-known high-level Petri net model, called *Predicate/Transition* (PrT) *nets*, was presented in 1979 by Genrich and Lautenbach (see [45]). It turned out that Predicate/Transition nets presented some technical problems when generalizing the invariant methods. To overcome this problem the *Coloured Petri Net* (CPN) model was defined in [69] by Jensen. For more information about the CPN model and the calculation of invariants in a high-level net, see Jensen et al. [69], [70], [71] and [72]. In theory it is also possible to extend a number of other analysis techniques to high-level nets. As long as the number of colours is finite, a high-level net is equivalent to a (much larger) Petri net without colours ('unfolding'). If the number of colours is infinite, then the high-

level net is equivalent to a basic Petri net with infinitely many places and transitions. Allowing an infinite number of colours results in a modelling power equivalent to a Turing machine for which many questions are undecidable (see Peterson [99]), but on the other hand, Church's thesis implies that the Turing machine is the most powerful model of computation (Wood [129]).

Our ITCPN model is a successor to the *DES* model developed by Van Hee, Somers and Voorhoeve ([53]). Like in the other high-level net models, a colour is attached to each token. Each place has a type (a set of colours) and tokens in a place have a colour (value) belonging to the corresponding type. The number of tokens produced by the firing of a transition, and their values (colours), may depend upon the values (colours) of the tokens consumed. Instead of using arc inscriptions, like in CPN, we use functions to describe the relation between the set of consumed tokens and the set of produced tokens. Note that, unlike in CPN, the enabling of a transition does not depend upon the values of the tokens to be consumed.

1.3.2 Adding time

The formal properties of 'Time' have attracted the attention of many philosophers, physicists and mathematicians (Benthem [14]). Time is an important aspect of all discrete dynamic systems. There are several ways to deal with this timing aspect.
First, one has to decide whether time has to be quantified. If time is not quantified, the model can only be used to reason about qualitative temporal properties, like liveness, mutual exclusion, deadlock, fairness, etc. We decide to quantify time, because only then, it is also possible to express quantitative temporal properties, like deadlines, activity durations, response times, delays, etc.
If time is quantified, one has to decide whether time is implicit or explicit. In physics and mathematics, time has traditionally been represented as just another variable. Consider for example first order predicate calculus, which can be used to reason about expressions containing a time variable, i.e. apparently there is no compelling need for explicit time. However, time plays a prominent part in the applications we consider, for we are interested in dynamic systems. Therefore, we decided to make time explicit (for reasons of convenience). This decision is based on the argument that the aspect of time is an important factor in the systems we want to consider, and the modelling effort is reduced considerably by adding explicit time constructs.

The basic Petri net model is not capable of handling quantitative time. The introduction of high-level nets allowed people to quantify time in an implicit manner, i.e. time is represented by the value or colour of a token. In this case, we have to model a global clock using a place connected to every transition. This place contains one token, whose value represents the current time. Since this is rather cumbersome, many authors have proposed a Petri net model with explicit quantitative time (e.g. [133], [108], [89], [82], [53], [113]). We call these models *Timed Petri Net* (TPN) models.

There are a lot of ways to introduce the concept of time into the basic Petri net model. In essence, there are two things one has to decide on: (1) the *location* of the time delays and (2) the *type* of these delays.

The location of the time delays

When introducing time into the basic Petri net model, we have to assign time durations (delays) to certain activities in the net. The literature on timed Petri nets describes many 'locations' in a Petri net which may be used to represent time.
Zuberek ([133]) associates a (fixed) delay with the firing time of a transition. When a transition fires, the enabling tokens are consumed and withheld for some time before the tokens appear in the output places. Since the firing of a transition takes some time, this is called 'two-phase' firing.
Sifakis and Wong propose models where time is associated with places, so that tokens arriving in a place are unavailable for a specified period ([114], [128]).
Most authors propose a model where time is associated with the enabling time of a transition (e.g. [41], [92], [82], [81]). Each transition in such a timed Petri net must remain enabled for a specified time before it can fire. In these models, firing is an atomic action, i.e. firing takes no time.
Some authors use two timing mechanisms (at different locations). An example of such mixture is the model proposed by Razouk and Phelps in [109], where time is associated with the firing of transitions and the enabling of transitions.

We use a rather new timing mechanism where time is associated with tokens. This timing concept has been adopted from Van Hee, Somers and Voorhoeve ([53]). In our ITCPN model we attach a *timestamp* to every token. This timestamp indicates the time a token becomes available. The *enabling time* of a transition is the maximum timestamp of the tokens to be consumed. Transitions are *eager* to fire (i.e. they fire as soon as possible), therefore the transition with the smallest enabling time will fire first. If, at any time, more than one transition is enabled, then any of these transitions may be 'the next' to fire. This leads to a non-deterministic choice if several transitions have the same enabling time. Firing is an atomic action, thereby producing tokens with a timestamp of at least the firing time. The difference between the firing time and the timestamp of such a produced token is called the *firing delay.*
Associating time with tokens is the logical choice for high-level Petri nets, since the colour is also associated with tokens. We will show that our timing concept is very expressive and allows for elegant semantics.

The type of the time delays

Besides the location of the delay, we also have to decide on the *type* of delay. There are three alternatives: fixed delays, stochastic delays or delays specified by an interval. We also have to decide whether we use a discrete or continuous time domain. Nearly all TPN models use a continuous time domain ($\mathbb{R}^+ \cup \{0\}$), so do we.

Petri nets with fixed (deterministic) delays have been proposed in [133], [108], [113] and [53]. They allow for simple analysis methods but are not very expressive.

In real discrete dynamic systems the duration of most activities is variable, because the duration of an activity often depends on external influences. Consider for example the time it takes to transport goods from a production unit to the central warehouse, this transportation time depends on traffic jams, the weather, the mood of the driver, etc. Clearly, a fixed delay is inappropriate for the modelling of the duration of such an activity.

One way to model this variability, is to assume certain delay distributions, i.e. to use a timed Petri net model with delays described by probability distributions. These nets are called *stochastic Petri nets*. Many stochastic Petri net models have been developed, most of them are used for the performance evaluation of protocols, manufacturing systems, etc. Two widespread models of this type are the SPN model by Florin and Natkin ([41]) and the GSPN model by Ajmone Marsan et al. ([82]). In nearly all stochastic TPN models, time is in transitions and the enabling time of such a transition is specified by some distribution. The choice of such a delay distribution is often difficult and subject to errors, thus yielding a crude approximation which appears to be exact.

Analysis of stochastic Petri nets is possible (in theory), since the reachability graph can be regarded, under certain conditions, as a Markov chain or a semi-Markov process. However, these conditions are severe: all firing delays have to be sampled from an exponential distribution or the topology of the net has to be of a special form (Ajmone Marsan et al. [81]). Since there are no general applicable analysis methods, several authors resorted to using simulation to study the behaviour of the net.

Another problem is the fact that the delays of two activities may be dependent. When modelling these activities by separate transitions, the delays are assumed to be independent, this may lead to incorrect results.

To avoid these problems, we propose delays described by an *interval* specifying an upper and lower bound for the duration of the corresponding activity. On the one hand, interval delays allow for the modelling of variable delays, on the other hand, it is not necessary to determine some artificial delay distribution (as opposed to stochastic delays). Instead, we have to specify bounds. These bounds can be used to verify time constraints. This is very important when modelling time-critical systems, i.e. *real-time* systems with 'hard' deadlines. These hard (real-time) deadlines have to be met for a safe operation of the system. An acceptable behaviour of the system depends not only on the logical correctness of the results, but also on the time at which the results are produced. Examples of such systems are: real-time computer systems, process controllers, communication systems, flexible manufacturing systems and just-in-time manufacturing systems.

To our knowledge, only one other model has been presented in literature which also uses delays specified by an interval. This model was presented by Merlin in [89] and [90]. In this model the enabling time of a transition is specified by a minimal and a maximal time. Another difference with our model is the fact that Merlin's model is not a high-level Petri net model because of the absence of typed (coloured) tokens. Compared to our model, Merlin's model has a rather complex formal semantics, which was presented in [16] by Berthomieu and Diaz. This is caused by a redundant state space (marking and enabled transitions are represented separately) and the fact that they use a relative time scale and allow for multiple enabledness of transitions. An additional advantage of our approach is the fact that our semantics closely correspond to our intuitive interpretation of the dynamical behaviour of a timed Petri net. We will motivate these statements in due time.

1.4 Analysis of timed coloured Petri nets

In the previous section we established the fact that Petri nets are appropriate for the modelling of discrete dynamic systems, provided that a Petri net model extended with time and colour is used. Based on this observation, we proposed the ITCPN model.
In essence, the modelling process serves two purposes. First of all, the model is used as a 'blueprint' of the system under consideration, e.g. the design of a new system or a plan which describes improvements. Secondly, models are used to analyse certain aspects of a system, e.g. the performance, efficiency or correctness of a system. Since analysis is often the main goal of model building, we have to supply suitable analysis methods.
In this section we start with a survey of existing analysis methods for timed and/or coloured Petri nets to illustrate that none of these methods (entirely) suits our purpose. This has been an incentive to develop new analysis methods. Therefore, the core of this monograph is directed towards the analysis of interval timed coloured Petri nets.

1.4.1 Currently used analysis methods

A lot of analysis techniques have been developed in the area of pure Petri net theory. Most of them are based on the basic Petri net model.
Many of these techniques have been extended to analyse high-level Petri nets, for example reachability graphs and invariants. Recall that as long as the number of colours is finite, a high-level net can be 'unfolded' into an equivalent, but much larger, Petri net without colours. The unfolding of nets has been studied to see how the analysis methods for high-level nets should work. For the moment, however, it is only possible to use these methods for relatively small systems and for selected parts of larger systems.
An example of such a method is the creation of a *reachability graph* for high-level nets. Because of the explosion of the number of states, these graphs tend to become

too large to analyse. Several reduction techniques have been proposed to deal with this problem. None of them gives a satisfactory solution (see Jensen [71]).

Another analysis technique available for high-level Petri nets is the generation of *place and transition invariants.* These invariants are used to derive and prove properties of the modelled system. A place invariant (P-invariant) is a weighted token sum, i.e. a weight is associated with every token in the net. This weight is based on the location (place) and the value (colour) of the token. A place invariant holds if the weighted token sum of all tokens remains constant during the execution of the net. Transition invariants (T-invariants) are the duals of place invariants and the basic idea behind them is to find firing sequences with no effects, i.e. firing sequences which reproduce the initial state. Some analysis techniques have been developed to calculate these invariants automatically (see Jensen [71]). These techniques have a number of problems. For large nets with a lot of different colours, it is hard to compute these invariants. Usually there are infinitely many invariants (a linear combination of invariants is also an invariant), therefore it is difficult to distill the interesting ones. However, there is a more promising way to use invariants. If the user supplies a number of invariants, it is easy to verify these invariants totally automatically. If an invariant does not hold, it is relatively easy to see how the Petri net (or the invariant) should be modified. The latter approach does not solve the problem that applying invariants requires a lot of training.

The addition of time to the basic Petri net model resulted in a lot of new and interesting techniques to analyse the dynamic behaviour of a system. Literature on this subject reflects the fact that the study of timed Petri nets developed along two separate lines.

The first line concentrates on the verification of dynamic properties. Most of the methods developed along this line are based on nets with deterministic delays. There are several methods to calculate upper and lower bounds for the *cycle time* of a timed Petri net ([113], [108], [107], [93]). The cycle time is a criterion for the performance of the system. For a specific class of deterministic timed Petri nets, the so-called Timed Event Graphs, the exact cycle time can be computed quite efficiently, see Ramamoorthy and Ho [107] and Chretienne [31]. Other researchers analyse deterministic timed Petri nets by building the reachability graph (Zuberek [133]). Although this requires a lot of computing effort, such a graph can be used to answer a variety of questions.

A serious drawback of these methods is the fact that in many real systems the activity durations are not fixed, i.e. they vary because of disturbances and other interferences. Assuming deterministic delays often results in inaccurate results.

The second line concentrates on the performance evaluation of timed Petri nets by means of analysis of the underlying stochastic process. Instead of assuming deterministic activity durations, an attempt is made to capture the essence of a system by probabilistic assumptions. These probabilistic assumptions often include

the distribution of the delays in the net. In nearly all stochastic TPN models a stochastic variable is associated with every transition. This stochastic variable expresses the delay from the enabling to the firing of a transition, i.e. the enabling time. For analysis reasons, the distribution of these stochastic variables is assumed to be negatively exponential. Molloy showed that, due to the memoryless property of the exponential distribution, such a stochastic TPN is isomorphic to a continuous time Markov chain ([92]). This allows for analytical methods to analyse the dynamic behaviour of a system, this way it is possible to calculate performance measures, e.g. the average waiting time or the probability of having more than five tokens in a specific place. Several other stochastic TPN models have been suggested ([82], [41], [80], [128], [64]). Consider for example, the Generalized Stochastic Petri Net (GSPN) model developed by Ajmone Marsan et al. ([82], [81], [83]). A GSPN has two types of transitions: 'timed' transitions and 'immediate' transitions. A timed transition has an exponentially distributed enabling time, an immediate transition has an enabling time of zero, i.e. an immediate transition fires the moment it becomes enabled.
Many authors give conditions for the topology of the net or the distribution of the delays such that analysis of the underlying stochastic process is possible (e.g. Ajmone Marsan et al. [81], [80]). In general these conditions are quite strong. Moreover, for real problems, the state space of the corresponding continuous time Markov chain tends to be too large to analyse.

To our knowledge, only one analysis method has been presented for Petri nets with interval timing. This method was presented by Berthomieu et al. in [17] and [16] and uses Merlin's timed Petri nets ([89]) to describe the system. The method generates a reachability graph where nodes represent state classes instead of states. This approach is more or less related to one of the analysis methods presented in this monograph.

Only a few analysis methods have been developed for timed *and* coloured Petri nets, this results from the fact that there are only a limited number of Petri net models having coloured tokens *and* some explicit time concept. In Lin and Marinescu [76] and Zenie [131] stochastic high-level nets are proposed. A high-level Petri net model with deterministic delays was presented by Van Hee et al. in [53]. A similar extension of the CPN model was proposed by Jensen in [71]. Note that a deterministic delay depending upon the colour of a token is sufficient to approximate any stochastic delay distribution, since coloured tokens allow for the generation of pseudo-random numbers, which can be used to sample delays for a specific distribution, see Shannon [112] or [9]. A straightforward way to analyse the dynamic behaviour of such a net is simulation.

1.4.2 Analysis methods based on the ITCPN model

Although Petri net theory is rich in analysis methods, only a few of the methods are suitable for the analysis of the temporal behaviour of a timed coloured Petri net.

Moreover, the methods used for the analysis of the dynamic behaviour of a system represented by a timed coloured Petri net suffer from computational problems. This is one of the reasons, simulation is the most widely used technique to analyse nets which represent complex discrete dynamic systems.

The ITCPN model deviates from existing models, because delays are specified by an interval rather than deterministic or stochastic delays. If we choose a distribution for each delay interval (e.g. a uniform or beta distribution), then we are able to simulate an ITCPN. Although simulation is a very powerful tool to analyse discrete dynamic systems, it is certainly not a panacea for answering all relevant questions. For example, simulation cannot be used to *prove* certain properties. This is one of the reasons, we have developed four analysis methods:

1. *Modified Transition System Reduction Technique* (MTSRT)
2. *Persistent Net Reduction Technique* (PNRT)
3. *Arrival Times in Conflict Free Nets* (ATCFN)
4. *Steady State Performance Analysis Technique* (SSPAT)

As said, these analysis methods are based on the ITCPN model.

The MTSRT method can be applied to any kind of ITCPN. This method generates a *reduced reachability graph*.
In an ordinary reachability graph, a node corresponds to a state. To calculate such an ordinary reachability graph, we start with an initial state, say s. For this state s, we obtain 'new states'. These are the states reachable by firing a transition in state s. New states are connected to s by a directed arc. For each new state, say s', connected to s, we obtain the states reachable by firing a transition in state s', etc. Repeating this process results in a graph representation of the reachable states.
Even for simple examples these graphs tend to be very large (generally infinite). The MTSRT method proposes a number of reductions, resulting in a reduced reachability graph. In such a graph a node corresponds to a set of states, called a *state class*, instead of a single state. To generate a graph representation of these state classes, we use a modified model, where a *time-interval* is associated with a token rather than a timestamp. We already mentioned a more or less related analysis method proposed by Berthomieu, Menache and Diaz in [17] and [16]. This method is based on Merlin's timed Petri net model. Their analysis method also uses state classes, which are represented by a system of inequalities. Our MTSRT method uses a totally different approach to analyse a Petri net with interval timing and is able to answer other types of questions. We will compare their method with our MTSRT method in due time.

The other methods can only be applied to a restricted set of interval timed coloured Petri nets.

The PNRT method and the SSPAT method can be applied to ITCPNs whose underlying net structure is a *marked graph*, i.e. the number of input arcs and output arcs of every place is smaller than or equal to 1. The PNRT method uses the special structure of such a net to create an even further reduced reachability graph. The SSPAT method calculates upper and lower bounds for the cycle time of a net. This is a generalization of the technique described by Ramamoorthy and Ho in [107].
The ATCFN method can be applied to conflict free nets, i.e. nets where the number of output arcs of every place is smaller than or equal to 1. This method produces upper and lower bounds for the arrival time of the first token in a place using a polynomial-time algorithm.

The analysis methods MTSRT, ATCFN and PNRT are outlined (in detail) in this thesis. For a description of the SSPAT method, see Van der Aalst [2].

For complex practical problems, the MTSRT method is most appropriate, because it can be applied to arbitrary interval timed coloured Petri nets. The conditions made by the other methods are often too restrictive. Furthermore, the MTSRT method is the only method able to answer questions involving the colour of tokens. The PNRT, ATCFN and the SSPAT abstract from the token colours. However, there are application areas where these limitations are not restrictive. For example: the ATCFN method can be used to analyse project plans, and the PNRT method can be used for production planning with repetitive schedules.

A consequence of the flexibility of the MTSRT method, is the computational effort required to analyse a complex system. For practical problems, the 'reduced' reachability graph generated by the MTSRT method, tends to become too large to analyse. In most cases this is caused by a large and complex net structure and/or a large number of possible token colours.
To deal with large colour sets, we propose techniques to translate an ITCPN into an ITCPN with only one kind of tokens, i.e. the cardinality of each colour set equals 1. Such an ITCPN is called an *Interval Timed Petri Net* (ITPN).
One can think of an ITPN as a specific kind of ITCPN with only one colour. Our aim, however, is to analyse interval timed coloured Petri nets. Therefore, we investigated suitable procedures for the translation of an ITCPN into an ITPN. There are two other reasons for having the desire to translate an ITCPN into an ITPN. First of all, ITCPNs with only one kind of tokens allow for several structural analysis techniques developed for uncoloured nets (see Murata [93]). Another reason is the fact that, at the moment, our analysis software only supports the analysis of uncoloured ITCPNs. Since we are able to (automatically) translate an ITCPN into an ITPN, we can analyse ITCPNs indirectly,

We distinguish three ways to translate an ITCPN into an ITPN:

unfold The first way is to translate the ITCPN into an equivalent ITPN is to use a construction similar to the one presented in Peterson [99] and Genrich [44].

Such a construction, often referred to as 'unfolding', is only possible if the number of colours is finite. The construction maps each place (transition) in the ITCPN into a set of places (transitions) in the constructed ITPN. If there are many different colours, the size of the constructed ITPN becomes very large. Therefore, this approach cannot be applied to large practical examples.

uncolour Another way to reduce the ITCPN into an ITPN is to discard the colours, to a certain extent. Each place in the ITCPN corresponds to exactly one place in the ITPN. If a transition in the ITCPN always produces the same number of tokens for every output place, then this transition also corresponds to exactly one transition in the ITPN. The lower bound (upper bound) of the delay of a token produced by a transition for a specific output place in the ITPN, corresponds to the smallest (largest) lower bound (upper bound) of all possible delays assigned to this place by the transition in the ITCPN. If the number of tokens produced by a transition in the ITCPN depends on the values of the consumed tokens, then this transition corresponds to a set of transitions in the ITPN. In practice the cardinality of this set is small. Therefore, this construction produces an ITPN of about the same size. Consider for example, a transition t with two output places o_1 and o_2. Assume that: if t fires, it produces one token, either for place o_1 or for place o_2 (depending upon the values of the consumed tokens). In the corresponding uncoloured net t is replaced by two transitions t_1 and t_2. Both transitions consume tokens from the input places of t. Transition t_1 produces a token for place o_1 and transition t_2 produces a token for place o_2.

Clearly some information is lost during this construction. However, it is still possible to derive useful properties for the ITCPN. For instance, if the ITPN is K-bounded (deadlock free), then the ITCPN is also K-bounded (deadlock free), and upper and lower bounds for the cycle time of the ITPN are also valid upper and lower bounds for the ITCPN. Often it is possible to *prove* certain properties for an ITCPN by analysing the corresponding ITPN, for example, it is possible to prove that certain deadlines are met.

refine The third way to use an ITPN to analyse an ITCPN is a mixture of the previous two. This hybrid approach works in two steps, first, for each place, the set of possible colours is partitioned into a number of colours sets, then the net is unfolded into an ITPN. A place in the ITCPN is mapped into a set of places, the cardinality of this set depends on the partitioning. In other words: first, we transform the ITCPN into an ITCPN with less colours and more places, then we remove the colours.

Consider for example an ITCPN with tokens representing machine jobs. The service time of a job depends on the colour of the token, i.e. its attributes. A job can have a large number of attributes, like weight, size, operations required, etc. In this case it is possible to partition the set of possible jobs into two meaningful classes: 'small' jobs and 'large' jobs. Based on this partitioning it is possible to derive upper and lower bounds for the service time of small (large) jobs. When unfolding the ITCPN into an ITPN, each place containing jobs is mapped into two places, one for small jobs and one for large jobs. The transitions connected to these places are also duplicated.

This way it is possible to derive tight bounds for the behaviour of the ITCPN without having an 'explosion' in the size of the net. Preferably, this approach is supported by a tool in an interactive way.

This monograph describes the last two approaches. These approaches are attractive, because they can be applied to *large coloured and timed* Petri nets, as opposed to nearly all other analysis methods. Note that this is a direct consequence of the fact that we use interval delays rather than deterministic or stochastic delays.

1.5 ExSpect

The practical use of the ITCPN model and related analysis methods highly depends upon the availability of adequate computer tools. To facilitate the creation, storage and adaptation of these models, we use a specification language to represent these models. We already mentioned that a formal specification is a precise and structured description of a system, expressed in a language having a syntax and semantics. We use the Petri net based specification language *ExSpect* ([53], [55], [52], [56], [51], [57], [8], [7]). This language has been developed at Eindhoven University of Technology, and is supported by a software package also called ExSpect (see Somers et al. [54], [9]).
We use ExSpect for the formal specification of a restricted class of interval timed coloured Petri nets. There is a straightforward relation between this specification language and the ITCPN model. In fact, the semantics of ExSpect are given in terms of a timed coloured Petri net model (see Van Hee et al. [53]).
The language ExSpect consists of two parts: a functional part and a dynamic part. The functional part is used to define types and functions needed to describe the operations on the value of a token. The type system consists of some primitive types and a few type constructors to define new types. A 'sugared lambda calculus' is used to define new functions from a set of primitive functions. ExSpect is a 'strongly typed' language since it allows all type checking to be done statically. A strong point of the language is the concept of type variables: it provides the possibility of polymorphic functions.
The dynamic part of ExSpect is used to specify a network of transitions and places, and therefore, the interaction structure of a system. The behaviour of a transition, i.e. the number of tokens produced and their values, is described by functions. The language also has a hierarchical construct called *system*. A system is a subnet, i.e. an aggregate of places and transitions and (perhaps) subsystems. The system concept supports both top-down and bottom-up design. A system can have a number of parameters. As a result, a system can be customized or fine-tuned for a specific situation. This way it is possible to define generic system specifications, that are easy to reuse.

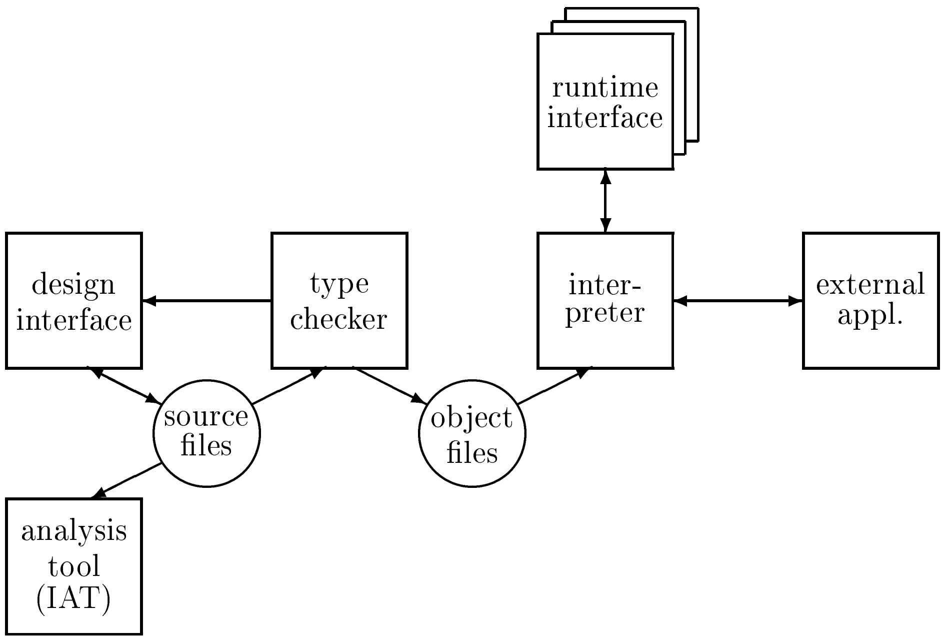


Figure 1.2: The toolset ExSpect

The software package ExSpect (EXecutable SPECification Tool) is a workbench based on the specification language ExSpect. This workbench is made up of a number of software tools, figure 1.2 shows the set of tools of ExSpect. These tools are integrated in a *shell*, from which the different tools can be started. The *design interface* is a graphical mouse driven editor, which is used to construct or to modify an ExSpect specification. Such a specification is stored in a source file (module). This source file is checked by the *type checker* for type correctness. If the specification is correct, then the type checker generates an object file, otherwise the errors are reported to the design interface. The *interpreter* uses the object file to execute a simulation experiment described by the corresponding ExSpect specification. This interpreter is connected to one or more *runtime interfaces*. These interfaces allow one or more users to interact with the running simulation. It is also possible to interact with some external application, for example presentation software.

Recently we added an analysis tool, called the *ITPN Analysis Tool* (IAT), to ExSpect. This tool translates a specification into an ITPN that is analysed using the methods described in this monograph, i.e. the MTSRT, PNRT and ATCFN analysis methods. The tool also allows for more traditional kinds of analysis such as the generation of P and T-invariants. This way we offer three kinds of analysis: simulation, 'structural analysis' (invariants) and 'interval analysis' (MTSRT, PNRT, ATCFN). This observation reveals an interesting issue: a formal specification can be used as a 'blueprint' of the system, which allows for various kinds of analysis.

This is very convenient, since it prevents us from having to remodel the system every time we want to use another analysis technique. Therefore, we are also interested in supporting other analysis techniques, e.g. Markovian analysis, queueing networks, linear programming, etc.

1.6 Application to logistics

High-level Petri nets have been used in many application areas: flexible manufacturing, computer architecture, distributed information systems, protocols, etc. In [72] there are a number of papers describing applications of high-level nets. We have used ExSpect in various application domains, e.g. queueing systems ([3]) and flexible manufacturing ([7]).

However, our main interest is in the modelling and analysis of logistic systems ([4], [5], [8], [6]). This interest stems from three reasons:
First of all, timed coloured Petri nets are an appropriate way to describe logistic processes. Note, that a logistic system is composed of physically distributed subsystems with a rather complex interaction structure, i.e. a typical example of a discrete dynamic system.
Secondly, recent developments in the field of logistics have complicated the management of the logistic processes, e.g. the integration of logistic activities often results in complex control problems. Therefore, there is a need for an integrated framework for the modelling and analysis of logistic systems.
Thirdly, we participate in a project called *TASTE* (The Advanced Studies of Transport in Europe). The goal of this project is to develop a tool to enable non-programmers to model and analyse strategic problems in the field of interindustrial logistics. TASTE uses ExSpect to model and analyse the flow of goods at an aggregated level in and between, production, assembly, distribution and transport (see [6]).

The TASTE project faced the fact that research in the field of logistics developed along two separate lines.
The first line concentrates on solving mathematical problems related to logistics. Investigations in this area are part of a discipline called *operations research.* Often the problem statement is simplified to allow for analytical solutions. This is the reason that many results in this area are not generally applicable and require an expert consultant. Examples of this line are the application of queueing networks to scheduling problems and the application of linear programming to transport planning. Although these analysis methods help us gain insight in the problem, they can only be applied in rather specific situations. Moreover, some of the results reported in this area describe techniques for problems that do not even exist in practice.
The second line of research concentrates on practical logistic problems. The results are often qualitative and informal. The approaches used in this area are mainly discipline oriented, i.e. they focus on a specific aspect of logistics. Examples are the

research on customer service, storage equipment, communication facilities (EDI), personnel requirements, etc.
Neither of these lines has lead to an integrated framework to model and analyse logistic systems. This is the reason this monograph outlines concepts and tools to facilitate the modelling and analysis of real logistic problems.
First, we motivate our choice to use timed coloured Petri nets. We will do this by showing that our Petri net model is able to represent typical logistic activities in a very convenient manner.
Secondly, we present a 'systems view of logistics' to structure complex logistic systems. Based on a taxonomy of the flows in a logistic system, we describe a systematic approach to the modelling of logistic systems. This approach can be used as a stepping-stone to the development of a comprehensive 'reference model' of logistics. Such a reference model is a representation of an idealized organization, defining the tasks of the logistic components as well as the interaction between these components (see Biemans et al. [19], [21]).
Thirdly, based on our 'systems view of logistics' we have developed an ExSpect library of predefined system definitions. These system definitions are parameterized *building blocks* representing typical logistic activities. There are about 20 of these building blocks including a production unit, a distribution centre and a transport system. It is our belief that many practical logistic systems can be modelled using these building blocks. Modelling in terms of building blocks is supported by software (ExSpect) and the modelling process results in a specification that can be analysed using simulation and the analysis methods already mentioned.

Our approach is intentionally *abstract*. Therefore, we focus on the main logistic functions (e.g. transport, demand, supply, production and stock holding) and ignore aspects, like administration, safety, personnel, etc. Moreover, sometimes we also abstract from the physical reality, i.e. we are not interested in the actual layout of a logistic system, mechanical aspects, communication protocols, etc.

1.7 Other methods

We use a Petri net based approach, this is only one of the many approaches which have been developed to model and analyse discrete dynamic systems. We distinguish three main directions:

- simulation techniques
- diagramming techniques
- formal techniques

Simulation is one of the most powerful techniques to analyse a complex system. Advantages of simulation are: easy to use, flexible, availability of tools. Another important advantage of simulation is that it helps the analyst to understand and

to gain a feel for the system. In a way, simulation is similar to the debugging of a program, in the sense that it can reveal errors of a (simulation) model. In practice, however, simulation is never sufficient to prove the correctness of the system.
There are two kinds of simulation tools: simulation languages and specific simulation packages. Simulation languages, such as SIMULA (Dahl and Nygaard [33]) and SIMAN (Pidd [103]), are flexible but lack sufficient support of the modelling process, e.g. a graphical editor, analysis tools, etc. Simulation packages are often application specific. Examples in the field of manufacturing are SIMFACTORY and TAYLOR ([103]). These packages are easy to use and support animation. The fact that they are tailored towards a specific application makes them inflexible. Note that, although ExSpect is a specification language, it can be used as a simulation language which can be tailored towards a specific domain by creating reusable systems, i.e. it is possible to use libraries of user-defined building blocks. The application of these building blocks is quite easy, because they can be used in a completely graphical manner.

There are several frameworks based on *diagramming techniques.* These frameworks use a graphical language to describe data flow, control flow, etc. The graphical nature of these frameworks makes them easy to use. Examples are SADT (Marca and McGowan [79]), ISAC (Lundeberg et al. [78]) and DFD (Ward and Mellor [121]). Most of these frameworks incorporate techniques to describe the data structure, for example the entity-relationship model (Chen [29]). The result of using such an approach is an informal description, that does not allow for quantitative analysis. Another drawback of these techniques is that they lack a concept to quantify time which makes it very difficult to model real-time constraints.

Formal methods to model (specify) and to analyse discrete dynamic systems are, at this point, mainly under development in the academic world. Some of these methods are slowly gaining industrial acceptance. We distinguish 6 directions:

- queueing networks
- finite state machines
- model oriented specifications
- process algebras
- temporal logic
- Petri nets

We will review these formal methods, without claiming to give a complete survey.

A *queueing network* (Ajmone Marsan et al. [83]) is a system of interconnected queues in which customers circulate, arrive or leave. Queueing networks have become quite popular in the field of performance evaluation. The main reason for this popularity is due to the product form solution, that holds for a restricted class of queueing networks (see Baskett et al. [13]). This restricted class allows for the analytical solution of all sorts of performance measures. Nevertheless, several practically important features, like synchronization, blocking and the splitting of customers can usually not be modelled in such a way that the model still has the product form solution (see Ajmone Marsan et al. [83]). For non-product form queueing networks there are approximative methods of analysis available, but these are not generally applicable and require an expert consultant. Therefore, for a more detailed analysis of queueing networks, simulation is practically unavoidable.

The *finite state machine* is a restriction to the classical model of theoretical computer science (Hopcroft and Ullman [66]). A finite state machine can be modelled using is a state transition diagram (Davis [34]). At any moment the machine is in a certain state. In response to an input the machine generates an output and changes state. *Statecharts* (Harel [48]) represents a generalized formalism based on finite state machines. In statecharts, the normal state transition diagram is enhanced with hierarchical and compositional features. Although a supporting tool, called 'statemate', has been developed, this method cannot be used to model large real-time systems because of the absence of facilities to model data structures and quantitative time.

VDM (Jones [73]) and *Z* (Spivey [116]) are *model oriented specification languages.* These methods have been found useful for the specification of large commercial systems, but are weak in their ability to deal with concurrency and real-time. Furthermore, these languages do not allow for quantitative analysis, the emphasis is on specification rather than analysis.

Process algebras, such as CSP (Hoare [63]), CCS (Milner [91]) and ACP (Bergstra and Klop [15]), are well suited for the modelling of parallel and concurrent behaviour. They are however poor in their capabilities to specify data structures and operations. There are several algebraic specification languages based on one of these process algebras, e.g. LOTOS (Brinksma [27], [26]) and PSF (Mauw and Veltink [86]). These languages have constructs to handle data structures, modularization and parameterization. Moreover, several process algebras have been extended with timing constraints, for example timed-CSP (Reed and Roscoe [110]), CCSR (Gerber and Lee [47]), ACPρ (Baeten and Bergstra [12]) and Timed LOTOS (Bolognesi et al. [23]).

Temporal logic (Pnueli [104]) is a branch of modal logic. Generally, a number of temporal operators are introduced, for example $\Box$ (henceforth) and $\Diamond$ (eventually). Various types of semantics can be given to the temporal operators depending on whether time is linear or branching, time is quantified, time is implicit or explicit, time is local or global, etc. A temporal logic is called a *real-time temporal logic* if time is quantified.

Metric Temporal Logic (Koymans [75]) is a real-time temporal logic with an implicit time construct. For example, the formula $A \rightarrow \Diamond_{\leq 3} B$ means that: if A occurs, then eventually within 3 time units B must occur.
Real-Time Temporal Logic (Ostroff [95]) has an explicit time (clock) variable t. The previous formula can be expressed as follows: $(A \wedge t = T) \rightarrow \Diamond(B \wedge t \leq T + 3)$.
An overview of existing frameworks in temporal logic is given by Ostroff in [96].
Temporal logic is suitable for describing (temporal) properties of a system. Disadvantages are the fact that temporal logic is difficult to learn and specifications based on temporal logic are hard to read. The low level nature of these specifications makes it difficult to model large and complex systems. Additional drawbacks are the absence of data modelling capabilities and limited analysis methods. A promising approach is the combination of temporal logic and other frameworks (e.g. Petri nets). Such an approach was presented by Ostroff in [95], where Extended State Machines are used to model the system and Real-Time Temporal Logic is used to specify the required behaviour of the system.

In this monograph we present an approach based on a timed coloured Petri net model. The Petri net concept meets the requirements set out by the distributed nature of a logistic system. The addition of colour and time, enables the modelling of data structures and a complex temporal behaviour. A major advantage compared to other methods mentioned in this section, is the availability of various kinds of analysis, e.g. simulation, 'structural analysis' (invariants) and 'interval analysis' (MTSRT, PNRT, ATCFN). From this point of view, this monograph provides an integrated approach which combines a number of existing formalisms.

1.8 Outline of this monograph

The remainder of this monograph consists of five chapters.
In Chapter 2 we define the ITCPN model. The semantics of this model is given in terms of a transition system. To do this, we introduce some basic notations and concepts. We also discuss some interesting properties of this model.
Chapter 3 describes three of the four analysis methods we have developed to analyse interval timed coloured Petri nets. These methods are compared with existing analysis methods. We also show how these methods can be used to analyse interval timed coloured Petri nets with large colour sets. We use an example to illustrate our approach.
In chapter 4 we discuss the language ExSpect and describe the tools that have been developed to support this language. The author participated in the development of the design interface and the analysis tool of ExSpect. As an example of an ExSpect module, we present the QNM library (see Van der Aalst [3]). This library contains building blocks, which can be used to model and analyse queueing networks in a graphical manner.
In chapter 5 we structure the field of logistics and discuss the application of Petri nets to logistic problems. We also present a library containing logistic building

blocks.
Finally, in chapter 6, we discuss the usefulness of the approach presented in this monograph.

Chapter 2

A timed coloured Petri net model

2.1 Introduction

In this chapter we give a formal definition of our ITCPN model. This chapter also describes some fundamental concepts, such as behavioural properties and performance measures. Some of these concepts have been adopted from existing Petri net theory, others have been developed with the rest of this monograph in mind. The concepts described in this chapter are used throughout this monograph and so they are fundamental to a correct understanding of our approach.

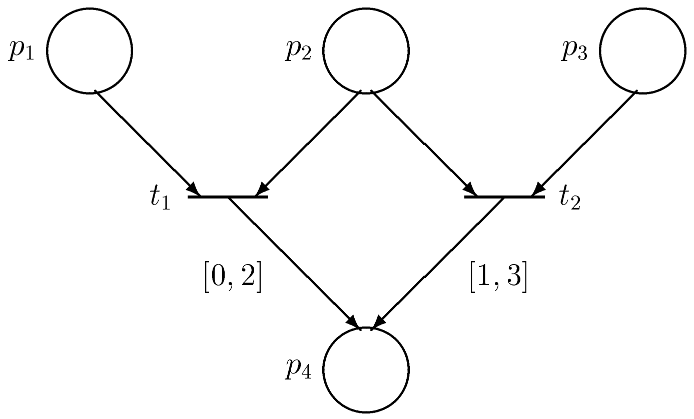


Figure 2.1: An interval timed coloured Petri net

In section 1.3 we already discussed the need for a timed and coloured Petri net model. This is the reason we developed the *Interval Timed Coloured Petri Net* (ITCPN) model.
To illustrate this model we use an example. Figure 2.1 shows an ITCPN which comprises four places (p_1, p_2, p_3 and p_4) and two transitions (t_1 and t_2). Transition t_1 has two input places (p_1 and p_2) and one output place (p_4). Transition t_2 also has

two input places (p_2 and p_3) and one output place (p_4). At any moment, a place contains zero or more tokens, drawn as black dots. In the ITCPN model, a token has four attributes: an identity, a position, a value and a timestamp, i.e. we can use the quartet $\langle i, p, v, x \rangle$ to denote a token in place p with value v, timestamp x and some identification number i.
Figure 2.2 shows the ITCPN in a state with one token in p_1, two tokens in p_2 and one token in p_3. In this example, the value of any token is a string, e.g. the token in place p_1 has a value $'AB'$. In the state shown in figure 2.2, both transitions t_1 and t_2 are enabled, because each of the input places of t_1 and t_2 contains at least one token. The enabling time of t_1 is the maximum timestamp of the tokens to be consumed, i.e. 3.0 (the maximum of 3.0 and 2.0). The enabling time of t_2 is 4.0 (the maximum of 2.0 and 4.0). Note that tokens on a place are consumed in order of their arrival (i.e. timestamps). Transitions are eager to fire, therefore t_1 fires at time 3.0.
Firing t_1 means consuming a token from place p_1 ($\langle 1, p_1, 'AB', 3.0 \rangle$) and place p_2 ($\langle 2, p_2, 'CD', 2.0 \rangle$) and producing a token for place p_4 whose value may depend on the values of the tokens consumed. In this case the value of the produced token is the concatenation of the values of the tokens consumed (i.e. $'ABCD'$). The delay of this token is between 0 and 2. Figure 2.3 shows a state resulting from the firing of transition t_1 in figure 2.2. In this case the delay of the token equals 1.25, however, any other value between 0 and 2 would have been allowed. The identification of the new token is an arbitrary, but unique, number (in this case 5).
In the state shown in figure 2.3 only t_2 is enabled. The enabling time of t_2 is 5.0 (the maximum of 5.0 and 4.0). Consequently, this transition fires at time 5.0. Transition t_2 also concatenates two strings, i.e. t_2 consumes a token from place p_2 ($\langle 3, p_2, 'EF', 5.0 \rangle$) and place p_3 ($\langle 4, p_3, 'GH', 4.0 \rangle$) and produces a token for place p_4 (e.g. $\langle 6, p_4, 'EFGH', 6.50 \rangle$). Note that in this case the delay of the produced token is 1.5.
Figure 2.4 shows a state resulting from the firing of transition t_2 in figure 2.3. There are no transitions enabled in this state.

The above example illustrates the dynamic behaviour of an ITCPN. It is, however, nearly impossible to give an informal explanation which is complete and unambiguous. Since an informal discussion of the meaning of interval timed coloured Petri nets is likely to cause confusion, we give a formal definition of the ITCPN model and the corresponding semantics in section 2.4.
Because our formalisms are based on bag theory and transition systems, we start with some useful notations and a formal definition of transition systems.

2.2 Notations

$\mathbb{N}$ is the set of natural numbers including zero. $\mathbb{R}$ is the set of reals. It is convenient to adjoin to $\mathbb{R}$ two additional elements, ∞ and $-\infty$ (not belonging to $\mathbb{R}$) with the order properties $-\infty < a < \infty$ for any $a \in \mathbb{R}$. We 'extend' the addition operator

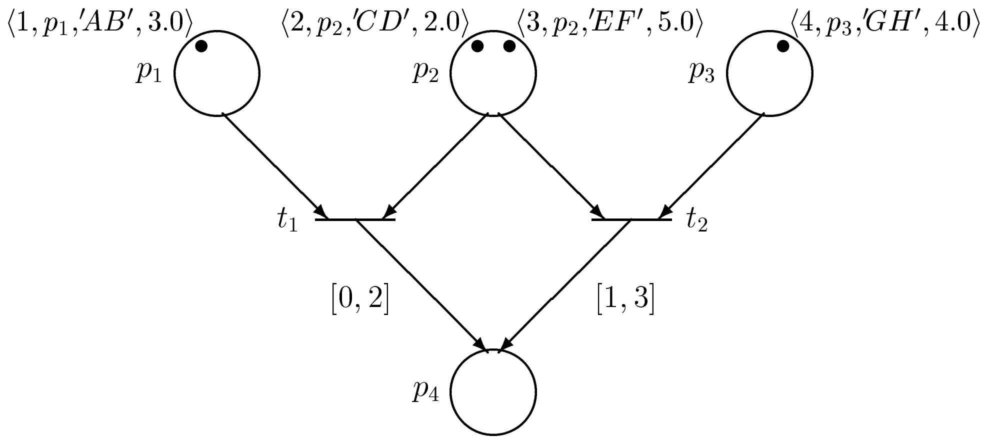

$\langle 1, p_1, 'AB', 3.0\rangle$
$\langle 2, p_2, 'CD', 2.0\rangle$
$\langle 3, p_2, 'EF', 5.0\rangle$
$\langle 4, p_3, 'GH', 4.0\rangle$
p_1
p_2
p_3
t_1
t_2
$[0,2]$
$[1,3]$
p_4

Figure 2.2: An ITCPN, t_1 and t_2 are enabled

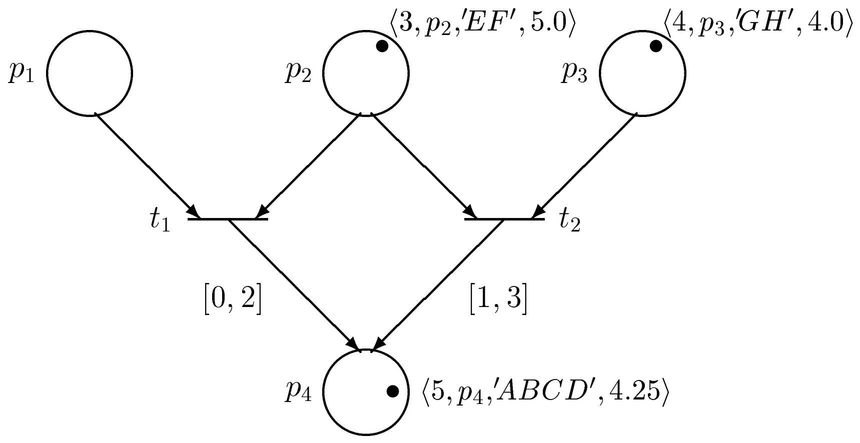

$\langle 3, p_2, 'EF', 5.0\rangle$
$\langle 4, p_3, 'GH', 4.0\rangle$
p_1
p_2
p_3
t_1
t_2
$[0,2]$
$[1,3]$
p_4
$\langle 5, p_4, 'ABCD', 4.25\rangle$

Figure 2.3: A state resulting from firing transition t_1

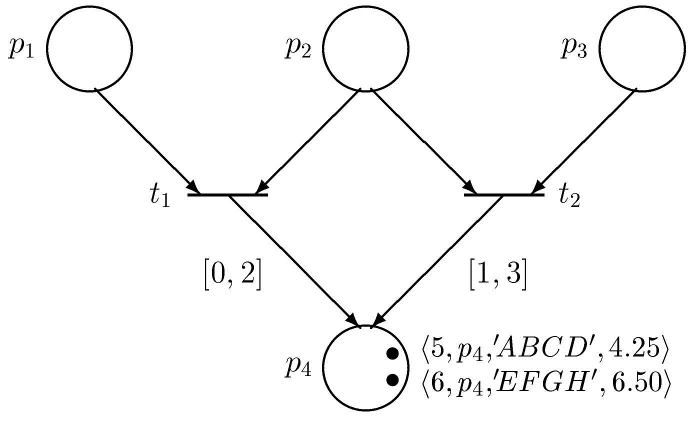

p_1
p_2
p_3
t_1
t_2
$[0,2]$
$[1,3]$
p_4
$\langle 5, p_4, 'ABCD', 4.25\rangle$
$\langle 6, p_4, 'EFGH', 6.50\rangle$

Figure 2.4: A state resulting from firing transition t_2

for reals such that for all $a \in \mathbb{R}$: $a + \infty = \infty + a = \infty$ and $\infty + \infty = \infty$. Similar conventions hold for $-\infty$. The expressions $\infty - \infty$ and $-\infty + \infty$ are undefined.

The Cartesian product of two sets A and B, denoted by $A \times B$, is the set of all ordered pairs $\langle a, b\rangle$ with $a \in A$ and $b \in B$. If $x = \langle a, b\rangle \in A \times B$ then $\pi_1(x) = a$ and $\pi_2(x) = b$. For $n \in \mathbb{N}$, $A_1, A_2, .., A_n$ sets, $x \in A_1 \times A_2 \times .. \times A_n$ and $i \in \{1, .., n\}$, $\pi_i(x)$ denotes the i^{th} component of x.

A binary relation R on a set S, is a subset of $S \times S$. If S a set and $R \subseteq S \times S$ then:
$R^0 = I = \{\langle s, s\rangle \mid s \in S\}$,
$R^n = \{\langle s_1, s_3\rangle \in S \times S \mid \exists_{s_2 \in S}(\langle s_1, s_2\rangle \in R \ \wedge \ \langle s_2, s_3\rangle \in R^{n-1})\}$, for $n > 0$ and
$R^* = \{\langle s_1, s_2\rangle \mid \exists_{n \in \mathbb{N}} \ \langle s_1, s_2\rangle \in R^n\} = \cup_{n \in \mathbb{N}} R^n$, the reflexive and transitive closure of R.

A partially ordered set, or just poset, is a pair $\langle S, R\rangle$ where S is a set and R a binary relation on S, which satisfies the following conditions:
$\forall_{s \in S} \ \langle s, s\rangle \in R$ (reflexive)
$\forall_{s_1, s_2 \in S} \ (\langle s_1, s_2\rangle \in R) \ \wedge \ (\langle s_2, s_1\rangle \in R) \ \Rightarrow \ (s_1 = s_2)$ (antisymmetric)
$\forall_{s_1, s_2, s_3 \in S} \ (\langle s_1, s_2\rangle \in R) \ \wedge \ (\langle s_2, s_3\rangle \in R) \ \Rightarrow \ (\langle s_1, s_3\rangle \in R)$ (transitive)

In general we denote a partial ordering by '$\leq$' and use an infix notation. We will adopt the notations $s_1 < s_2$, $s_1 \geq s_2$, $s_1 > s_2$ for respectively $s_1 \leq s_2 \ \wedge \ s_1 \neq s_2$, $s_2 \leq s_1$, $s_2 \leq s_1 \ \wedge \ s_1 \neq s_2$. A poset $\langle S, \leq\rangle$ is a linear ordering (total ordering), if and only if, for all $s_1, s_2 \in S$: $s_1 \leq s_2$ or $s_2 \leq s_1$.

Set operations are defined in the usual way. If A is a set, then #A is the number of elements in A and $\mathbb{P}(A)$ is the powerset of A (the set of all subsets of A).

For A and B sets, $A \rightarrow B$ denotes the set of all total functions from A to B and $A \not\rightarrow B$ denotes the set of all partial functions from A to B.

If $f \in A \not\rightarrow B$ then $dom(f)$ is the domain of f and $rng(f) = \{f(x) \mid x \in dom(f)\}$ is the range of f.

If f a function then f is also defined for $X \subseteq dom(f)$: $f(X) = \{f(x) \mid x \in X\}$.

$f \restriction X$ denotes the restriction of a function to $X \subseteq dom(f)$, i.e. $dom(f \restriction X) = X$ and for all $x \in X$: $f \restriction X(x) = f(x)$.

We use the lambda notation or the 'set notation' to define functions, i.e. a function $f = \lambda_{x \in dom(f)} f(x) = \{\langle x, f(x)\rangle \mid x \in dom(f)\}$.

Note that the set notation of a function allows for a number of set operations. If f_1, f_2 are functions, then:
$\#f_1 = \#dom(f_1)$
$f_1 \subseteq f_2$ iff $dom(f_1) \subseteq dom(f_2) \ \wedge \ \forall_{x \in dom(f_1)} \ f_1(x) = f_2(x)$
$f_1 \setminus f_2 = f_1 \restriction \{x \in dom(f_1) \mid x \in dom(f_2) \ \Rightarrow \ f_1(x) = f_2(x)\}$

Furthermore, if f_1, f_2 functions with disjoint domains then:
$f_1 \cup f_2 = \{\langle x, y\rangle \mid (x \in dom(f_1) \;\wedge\; f_1(x) = y) \;\vee\; (x \in dom(f_2) \;\wedge\; f_2(x) = y)\}$

For a totally ordered set A and $x, y \in A$: x min y (x max y) is the minimum (maximum) of x and y, i.e. if $x \leq y$ then x min $y = x$ (x max $y = y$). If A is a totally ordered finite non-empty set, then min A is the minimal element of A and max A is the maximal element of A. If $A = \emptyset$, then min $A = \infty$ and max $A = -\infty$. If $A \subseteq \mathbb{R} \cup \{-\infty, \infty\}$ then min A (max A) is the supremum (infimum) of A. If A is not bounded below (above) then min $A = -\infty$ (max $A = \infty$). Because of the completeness axiom for reals (see Depree and Swartz [36]), every subset of $\mathbb{R} \cup \{-\infty, \infty\}$ has a supremum and infimum.
Sometimes we use an alternative notation to denote the minimum (maximum) of the range of a function f on a specified domain A: $\min_{x \in A} f(x) = \min\{f(x) \mid x \in A\}$ and $\max_{x \in A} f(x) = \max\{f(x) \mid x \in A\}$.

Intuitively a *multiset* is the same as a set, except for the fact that a multiset may contain multiple occurrences of the same element. Another word for multiset is *bag*. Bag theory is a natural extension of set theory (see Peterson [100]). A multiset, like a set, is a collection of elements over the same subset of some universe. However, unlike a set, a multiset allows multiple occurrences of the same element. A multiset b over A is defined by a function from A to $\mathbb{N}$, i.e. $b \in A \rightarrow \mathbb{N}$. If $a \in A$ then $b(a)$ is the number of occurrences of a in the multiset b. $\mathbb{B}(A)$ is the set of all multisets over A.
We now introduce some operations on bags. Most of the set operators can be extended to bags in a rather straightforward way. Suppose A a set, $b_1, b_2 \in \mathbb{B}(A)$ and $q \in A$.

$q \in b_1$ iff $b_1(q) \geq 1$	(membership)
$b_1 \subseteq b_2$ iff $\forall_{a \in A}\; b_1(a) \leq b_2(a)$	(inclusion)
$b_1 = b_2$ iff $b_1 \subseteq b_2 \;\wedge\; b_2 \subseteq b_1$	(equality)
$b_1 \cup b_2 = \lambda_{a \in A}\;\; (b_1(a) \max b_2(a))$	(union)
$b_1 \cap b_2 = \lambda_{a \in A}\;\; (b_1(a) \min b_2(a))$	(intersection)
$b_1 + b_2 = \lambda_{a \in A}\;\; (b_1(a) + b_2(a))$	(sum)
$b_1 \setminus b_2 = \lambda_{a \in A}\;\; ((b_1(a) - b_2(a)) \max 0)$	(difference)
$\min(b_1) = \min\{a \in A \mid a \in b_1\}$	(minimum)
$\max(b_1) = \max\{a \in A \mid a \in b_1\}$	(maximum)
$\#b_1 = \sum_{a \in A} b_1(a)$	(cardinality of a finite bag)

We use square brackets to denote multisets by enumeration. Suppose A a set, $n \in \mathbb{N}$ and $q_0, q_1, .., q_n \in A$ then $[q_0, q_1, .., q_n] = \lambda_{a \in A}\; \#\{i \in \{0, .., n\} \mid q_i = a\}$. Consider, for example, the following bags over the domain $\mathbb{N}$: $[1, 3]$, $[1, 1, 1]$, $[1, 2, 1, 2]$. Note that $[1, 2, 1, 2]$ and $[1, 1, 2, 2]$ indicate the same bag. We use $[\;]$ to denote the empty bag.

Although bags are a generalization of sets, we want to be able to represent bags as sets. This can be done by attaching a unique label to every element in the bag. An advantage of such a *labelled bag* is the fact that it is possible to identify single elements in a bag. In the rest of this monograph we assume that there is an infinite set of labels called Id, for example $Id = \mathbb{N}$. More formally: we represent a finite bag $b \in \mathbb{B}(A)$ by a partial function $s \in Id \not\rightarrow A$ with a finite domain. In order to be able to switch between the two types of representation, we introduce two conversion functions: $\mathcal{SB}$ and $\mathcal{BS}$.

Definition 1
If A is a set then we define $\mathcal{SB} \in (Id \not\rightarrow A) \not\rightarrow \mathbb{B}(A)$ and a $\mathcal{BS} \in \mathbb{B}(A) \not\rightarrow (Id \not\rightarrow A)$ as follows. For any $s \in Id \not\rightarrow A$ with a finite domain and for any finite bag $b \in \mathbb{B}(A)$, we have:

$$\mathcal{SB}(s) = \lambda_{a \in A} \#\{i \in \mathit{dom}(s) \mid s(i) = a\}$$
$$\mathcal{SB}(\mathcal{BS}(b)) = b$$

Function $\mathcal{SB}$ transforms a labelled bag into the conventional representation without labels. Note that several functions $\mathcal{BS}$ satisfying the condition $\forall_{b \in \mathbb{B}(A)} \mathcal{SB}(\mathcal{BS}(b)) = b$ are possible ('Axiom of Choice'). It is easy to verify that such a function exists, e.g. take one element from the bag and label it 1, take an arbitrary other one and label it 2, etc. For example, if A is a totally ordered set and $Id = \mathbb{N}$, then we may define $\mathcal{BS}$ as follows. For any finite $b \in \mathbb{B}(A)$: $\mathcal{BS}(b) = \mathit{label}(Id, b)$, where for any $X \subseteq Id$:

$$\mathit{label}(X, b) = \begin{cases} \emptyset & \text{if } b = [\,] \\ \{\langle \min X, \min b \rangle\} \cup \mathit{label}(X \setminus \{\min X\}, b \setminus [\min b]) & \text{if } b \neq [\,] \end{cases}$$

In the remainder of this monograph we assume a given $\mathcal{BS}$, i.e. a fixed function.

Definition 2
Two labelled bags over A, say $s_1, s_2 \in Id \not\rightarrow A$, are equal if and only if the corresponding bags are equal, i.e. $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$.

If two labelled bags are equal, then there is an obvious bijection between the elements. This is expressed by the following lemma:

Lemma 1
Let A be a set and $s_1, s_2 \in Id \not\rightarrow A$. Then $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$ if and only if there exists a bijective function $f \in \mathit{dom}(s_1) \rightarrow \mathit{dom}(s_2)$ with:

$$\forall_{i \in \mathit{dom}(s_1)}\ s_1(i) = s_2(f(i))$$

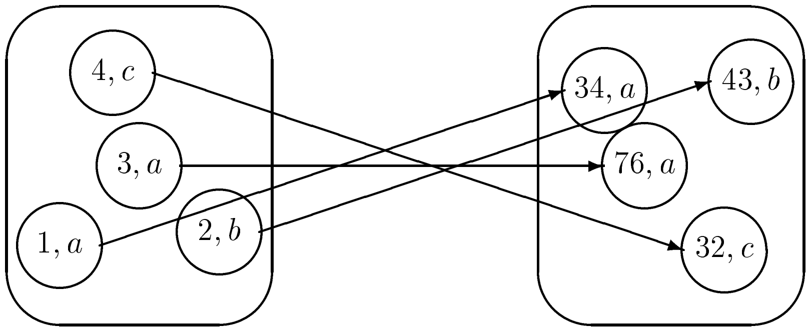


Figure 2.5: Two equivalent labelled bags

Proof.
Let $s_1, s_2 \in Id \not\rightarrow A$.

(1) Assume that there exists a bijective $f \in dom(s_1) \rightarrow dom(s_2)$ with for all $i \in dom(s_1)$: $s_1(i) = s_2(f(i))$. Now we have to prove that $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$.
For any $a \in A$: $\{i \in dom(s_1) \mid s_1(i) = a\} = \{i \in dom(s_1) \mid s_2(f(i)) = a\}$ and
$\#\{i \in dom(s_1) \mid s_2(f(i)) = a\} = \#\{j \in dom(s_2) \mid s_2(j) = a\}$ (f is bijective).
Hence, $\lambda_{a \in A}\#\{i \in dom(s_1) \mid s_1(i) = a\} = \lambda_{a \in A}\#\{j \in dom(s_2) \mid s_2(j) = a\}$, i.e. $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$.

(2) Assume that $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$. Now we have to prove that there exists a bijective $f \in dom(s_1) \rightarrow dom(s_2)$ with for all $i \in dom(s_1)$: $s_1(i) = s_2(f(i))$.
For any $a \in A$: $\#\{i \in dom(s_1) \mid s_1(i) = a\} = \#\{j \in dom(s_2) \mid s_2(j) = a\}$, because $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$.
If X and Y two arbitrary sets and $\#X = \#Y$, then there exists a bijective $g \in X \rightarrow Y$. Hence, for each $a \in A$, there exists a bijective function $f_a \in \{i \in dom(s_1) \mid s_1(i) = a\} \rightarrow \{j \in dom(s_2) \mid s_2(j) = a\}$.
If $a_1, a_2 \in A$ and $a_1 \neq a_2$, then $dom(f_{a_1}) \cap dom(f_{a_2}) = \emptyset$ and $rng(f_{a_1}) \cap rng(f_{a_2}) = \emptyset$.
Consequently, $f = \cup_{a \in A} f_a$ is bijective and for all $i \in dom(s_1)$: $s_1(i) = s_2(f(i))$.
$\square$

Figure 2.5 shows a bijective function f relating two equivalent labelled bags. In this case, $dom(f) = \{1, 2, 3, 4\}$, $f(1) = 34$, $f(2) = 43$, $f(3) = 76$ and $f(4) = 32$.

2.3 Transition systems

To formalize the ITCPN model we have to attach a precise meaning to interval timed coloured Petri nets, this can be done by giving formal semantics. There are several ways to do this. In literature three styles of semantics are distinguished: (1) operational semantics, (2) axiomatic semantics and (3) denotational semantics. We

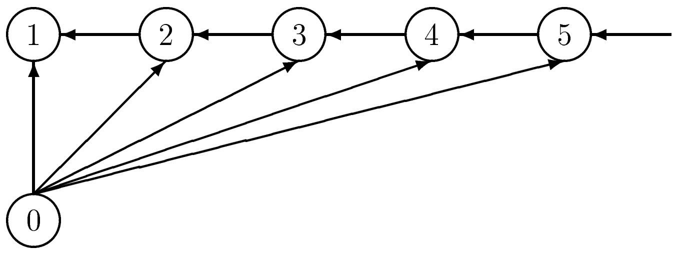


Figure 2.6: A graphical representation of $\langle S, R\rangle$

use operational semantics [1] to describe our formalism, because this seems to be the most natural way to describe the behaviour of an interval timed coloured Petri net. Another advantage of using operational semantics is that it makes it easy to compare two models by establishing a relation between the states of the two models. We use this property to prove the correctness of some of the analysis methods described in chapter 3.
The operational semantics (of the behaviour) of our model are given by means of a *transition system.* There are several types of transition systems, called labelled transition systems, non-deterministic machines, process graphs, non-deterministic automata, etc. (see Milner [91], Hennessy [59], Hesselink [60], Van Hee and Rambags [49], etc.). We define a transition system as follows:

Definition 3 (Transition System)
A transition system is a pair $\langle S, R\rangle$, where:

S is a set	, called the *state space*
$R \subseteq S \times S$	, the *transition relation*

A similar definition is given by Van Hee and Rambags in [49]. Note that actions, i.e. transitions from one state to another, are not labelled as opposed to many existing types of transition systems. Although our definition deviates from most transition systems described in literature (e.g. Hesselink [60]), we use definition 3 for reasons of convenience. Furthermore, it is easy to transform our transition systems into any other type of transition systems and vice versa.

Sometimes it is useful to make a graphical representation of a transition system. Consider for example the transition system $\langle S, R\rangle$, where:
$S = \mathbb{N}$
$R = \{\langle n+2, n+1\rangle \mid n \in \mathbb{N}\} \ \cup\ \{\langle 0, n\rangle \mid n \in \mathbb{N} \setminus \{0\}\}$
The corresponding graph is shown in figure 2.6.

Reachability is the basis for studying the behaviour of a transition system.

[1]In a sense, our semantics are also denotational semantics, since we specify the meaning of an ITCPN by mathematical objects, such as sets, functions and relations.

Definition 4 (Reachability)
For a transition system $\langle S, R \rangle$ and an initial state $s \in S$ we define:
$R(s) = \{\hat{s} \in S \mid sR\hat{s}\}$, the *one step reachability set* of s
$R^n(s) = \{\hat{s} \in S \mid sR^n\hat{s}\}$, the *n-step reachability set* of s
$RS(s) = \cup_{n \in \mathbb{N}} R^n(s)$, the set of all states that are reachable from s
$S^T = \{\hat{s} \in S \mid R(\hat{s}) = \emptyset\}$, the set of *terminal states*

For the transition system depicted in figure 2.6, $R(0) = \mathbb{N} \setminus \{0\}$, $R(1) = \emptyset$, $R(2) = \{1\}$, $R(3) = \{2\}$, $R^2(0) = \mathbb{N} \setminus \{0\}$, $R^2(1) = \emptyset$, $R^2(2) = \emptyset$, $R^2(3) = \{1\}$, $RS(0) = \mathbb{N}$ and for $n > 0$: $RS(n) = \{k \in \mathbb{N} \mid 1 \leq k \leq n\}$. Note that state 1 is a terminal state.

The *process* of a transition system starting in an initial state s is described by the set of all possible *execution paths* starting in s. These execution paths represent all possible 'behaviours' of the transition system. An execution path is a (maximal) sequence of states such that any successive pair belongs to the transition relation. A path starts in an initial state and either it is infinite or it ends in a terminal state.

Definition 5 (Process)
For a transition system $\langle S, R \rangle$ and an initial state $s \in S$ we define:

$$\begin{aligned} \Pi(s) = \{\sigma \in \mathbb{N} \not\rightarrow S \quad | \quad & 0 \in dom(\sigma) \ \wedge \ \sigma_0 = s \\ & \wedge \ \forall_{i \in dom(\sigma) \setminus \{0\}} \ (i-1) \in dom(\sigma) \ \wedge \ \sigma_{i-1} R \sigma_i \\ & \wedge \ \forall_{i \in dom(\sigma)} \ (\forall_{j \in dom(\sigma)} \ j \leq i) \Rightarrow \sigma_i \in S^T \ \} \end{aligned}$$

$\Pi(s)$ is the process (or behaviour) of the transition system in state s.

Note that the domain of a firing sequence σ is consecutive subset of $\mathbb{N}$. Consider the transition system shown in figure 2.6. Examples of paths starting in state 0 are $\{\langle 0,0\rangle, \langle 1,1\rangle\}$, $\{\langle 0,0\rangle, \langle 1,2\rangle, \langle 2,1\rangle\}$ and $\{\langle 0,0\rangle, \langle 1,4\rangle, \langle 2,3\rangle, \langle 3,2\rangle, \langle 4,1\rangle\}$. $\Pi(s)$ is the set of all possible execution paths starting in s. For all paths $\sigma \in \Pi(s)$ and $n \in \mathbb{N}$: $\sigma \restriction \{k \in \mathbb{N} \mid 0 \leq k < n\}$ is called a *trace*.

One of the main reasons for choosing operational semantics is the fact that it allows us to compare the behaviour of two systems. Therefore, we introduce some concepts to compare transition systems. Most of these concepts have been adopted from Hesselink [60] and Van Hee and Rambags [49].

The first relationship we consider is the so-called morphism from one transition system to another.

Definition 6 (Morphism)
Let $X = \langle S_x, R_x \rangle$ and $Y = \langle S_y, R_y \rangle$ be two transition systems. A function $f \in S_x \rightarrow S_y$ is a morphism from transition system X to transition system Y if and only if:

$$\{\langle f(x_1), f(x_2) \rangle \mid \langle x_1, x_2 \rangle \in R_x\} \subseteq R_y$$

Loosely speaking, a function f is called a morphism from transition system X to transition system Y if every transition in X corresponds to some transition in Y. The morphism is said to be *strict* if:

$$\{\langle f(x_1), f(x_2)\rangle \mid \langle x_1, x_2\rangle \in R_x\} = R_y$$

It is easy to verify that the composition of morphisms is transitive:

Lemma 2
Let $X = \langle S_x, R_x\rangle$, $Y = \langle S_y, R_y\rangle$ and $Z = \langle S_z, R_z\rangle$ be transition systems. If $f \in S_x \rightarrow S_y$ is a morphism from transition system X to transition system Y and $g \in S_y \rightarrow S_z$ is a morphism from transition system Y to transition system Z, then $g \circ f \in S_x \rightarrow S_z$ is a morphism from X to Z.

Proof.
Straightforward.
□
If both morphisms are strict, then so is the composition.

Sometimes it is not possible to establish a functional relationship between two transition systems. Consider for example two transition systems X and Y where one state in X corresponds to two or more states in Y *and* vice versa. In this case we are in need of a weaker relationship. This relationship is called *similarity*, it is based on a relation rather than a function.

Definition 7 (Similarity)
Let $X = \langle S_x, R_x\rangle$ and $Y = \langle S_y, R_y\rangle$ be two transition systems. Y is similar to X with respect to a relation $C \subseteq S_x \times S_y$ if and only if:

$$\forall_{\langle x_1,x_2\rangle \in R_x}\ \forall_{y_1 \in S_y}\ (\langle x_1, y_1\rangle \in C) \ \Rightarrow\ \exists_{y_2 \in S_y}(\langle x_2, y_2\rangle \in C \ \wedge\ \langle y_1, y_2\rangle \in R_y)$$

This definition is illustrated by figure 2.7. For every transition $\langle x_1, x_2\rangle$ in X and every state y_1 in Y related to x_1 (i.e. $\langle x_1, y_1\rangle \in C$), there exists a transition from y_1 to a state y_2 such that y_2 is related to x_2. To clarify this concept, consider the following example: $X = \langle S_x, R_x\rangle$ and $Y = \langle S_y, R_y\rangle$ are two transition systems defined as follows:

$$\begin{aligned}
S_x &= \mathbb{N} \\
R_x &= \{\langle n, n+1\rangle \mid n \in \mathbb{N}\} \\
S_y &= \{\langle k, l\rangle \mid k \in \mathbb{N} \ \wedge\ l \in \mathbb{N} \ \wedge\ k \leq l\} \\
R_y &= \{\langle\langle k, l\rangle, \langle k+1, l+1\rangle\rangle \mid \langle k, l\rangle \in S_y\} \\
C &= \{\langle n, \langle k, l\rangle\rangle \in S_x \times S_y \mid k \leq n \leq l\}
\end{aligned}$$

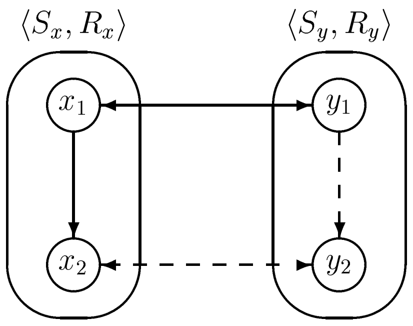


Figure 2.7: The 'similarity' relationship

It is easy to verify that Y is similar to X with respect to C. The 'soundness' and 'completeness' properties defined in chapter 3 are also examples of similarity relations.

The composition of similarity relations is transitive.

Lemma 3
Let $X = \langle S_x, R_x \rangle$, $Y = \langle S_y, R_y \rangle$ and $Z = \langle S_z, R_z \rangle$ be transition systems. If Y is similar to X with respect to a relation $C_1 \in S_x \times S_y$ and Z is similar to Y with respect to a relation $C_2 \in S_y \times S_z$, then Z is similar to X with respect to the relation:

$$C = \{\langle x, z \rangle \in S_x \times S_z \mid \exists_{y \in S_y} \ \langle x, y \rangle \in C_1 \ \wedge \ \langle y, z \rangle \in C_2\}$$

Proof.
Straightforward.
□

A morphism of two transition systems is a special form of similarity.

Lemma 4
Let $X = \langle S_x, R_x \rangle$ and $Y = \langle S_y, R_y \rangle$ be transition systems. If $f \in S_x \rightarrow S_y$ is a morphism from X to Y, then Y is similar to X with respect to a relation $C = \{\langle x, f(x) \rangle \mid x \in S_x\}$.

Sometimes a similarity relation is bidirectional. Consider the previous example, Y is similar to X with respect to $C = \{\langle n, \langle k, l \rangle \rangle \in S_x \times S_y \mid k \leq n \leq l\}$ and X is similar to Y with respect to $\hat{C} = \{\langle \langle k, l \rangle, n \rangle \in S_y \times S_x \mid k \leq n \leq l\}$. Therefore, many authors define a concept called *bisimilarity* (e.g. Hesselink [60]).

Definition 8 (Bisimilarity)
Let $X = \langle S_x, R_x \rangle$ and $Y = \langle S_y, R_y \rangle$ be two transition systems. X and Y are said to be bisimilar with respect to a relation $C \subseteq S_x \times S_y$, if and only if, Y is similar to X with respect to C and X is similar to Y with respect to $\{\langle y, x \rangle \mid \langle x, y \rangle \in C\}$.

It is easy to see that bisimilarity is reflexive, symmetric and transitive, i.e. an equivalence relation. Note that for any transition system X and Y, X and Y are bisimilar with respect to $C = \emptyset$. Therefore, we introduce a stronger relationship, called *equivalence.*

Definition 9 (Equivalence)
Let $X = \langle S_x, R_x \rangle$ and $Y = \langle S_y, R_y \rangle$ be two transition systems. X and Y are said to be equivalent, if and only if, there exists a strict bijective morphism $f \in S_x \to S_y$ from X to Y.

Function f in definition 9, is called an *isomorphism* from X to Y (and vice versa). If two transition systems X and Y are equivalent there is a one-to-one correspondence between the states of X and Y. A transition between two states of X is possible if and only if the corresponding transition is possible in Y, i.e $x_1 R_x x_2 \Rightarrow f(x_1) R_y f(x_2)$ and $y_1 R_y y_2 \Rightarrow f^{-1}(y_1) R_x f^{-1}(y_2)$. Using lemma 4 it is easy to verify that the equivalence of X and Y implies that X and Y are bisimilar with respect to relation $C = \{\langle x, f(x) \rangle \mid x \in S_x\}$.
This completes our introduction to transition systems.

2.4 The model

An *interval timed coloured Petri net* (ITCPN) is a directed labelled bipartite graph with two node types called *places* and *transitions.* Places are represented by circles and transitions by bars. A directed *arc* (arrow) connects a place and a transition in only one direction. A place p is called an *input place* of a transition t if there exists a directed arc from p to t. A place p is called an *output place* of a transition t if there exists a directed arc from t to p. Places may contain zero or more *tokens*, drawn as black dots. The number of tokens may change during the execution of the net. The place where a token 'resides' is called the *position* (or location) of a token. Besides a position, a token also has a *value*, a *timestamp* and some *identification.*
The timestamp indicates the time the token becomes *available.* The identification is merely used to discriminate between two tokens having an identical value and timestamp.
A transition is called *enabled* if there are 'enough' tokens on each of its input places. In other words, a transition is enabled if all input places contain (at least) the specified number of tokens. An enabled transition can *fire* at time x if all the tokens to be consumed have a timestamp not later than time x. The *enabling time* of a transition is the maximum timestamp of the tokens to be consumed. Because transitions are eager to fire, a transition with the smallest enabling time will fire first.
Firing a transition means consuming tokens from the input places and producing tokens on the output places. If, at any time, more than one transition is enabled, then any of the several enabled transitions may be 'the next' to fire. This leads to a non-deterministic choice if several transitions have the same enabling time.

Firing is an atomic action, thereby producing tokens with a timestamp of at least the firing time. The difference between the firing time and the timestamp of such a produced token is called the *firing delay*. This delay is specified by an *interval*, i.e. only delays between a given upper bound and a given lower bound are allowed. In other words, the delay of a token is 'sampled' from the corresponding delay interval. Note that the term 'sampled' may be confusing, because the modeller does not specify a probability distribution, merely an upper and lower bound. Moreover, it is possible that the modeller specifies a delay interval which is too wide, because of a lack of detailed information. In this case, the actual delays (in the real system) only range over a part of the delay interval.
The number of tokens produced by the firing of a transition may depend upon the values of the consumed tokens. Moreover, the values and delays of the produced tokens may also depend upon the values of the consumed tokens. The relation between the values of the consumed tokens and the bag of produced tokens is described by a function. Note that, unlike in CPN, the enabling of a transition does not depend upon the values of the tokens consumed.

Definition 10 (ITCPN)
An ITCPN is defined by a seven tuple, ITCPN $= (P, V, T, I, O, F, TS)$ with:

- $P = dom(V)$, the set of *places*
- V is a function with domain P, for all $p \in P$:
  V_p is the *value set* or *colour set* of p ($V_p \neq \emptyset$)
- $T = dom(I) = dom(O) = dom(F)$, the set of *transitions*
- $I \in T \rightarrow \mathbb{B}(P)$, the *input places* of a transition and their weights
- $O \in T \rightarrow \mathbb{P}(P)$, the *output places* of a transition
- TS, the *time set*
- $INT = \{\langle t_1, t_2 \rangle \in TS \times TS \mid t_1 \leq t_2 \land t_1 < \infty\}$, the set of all possible closed *intervals*
- $CT = \{\langle p, v \rangle \mid p \in P \land v \in V_p\}$, the set of all possible *coloured tokens*
- F is the *transition function*, for all $t \in T$, $F_t \in \mathbb{B}(CT) \not\rightarrow \mathbb{B}(CT \times INT)$, such that:

  $$dom(F_t) = \{c \in \mathbb{B}(CT) \mid \forall_{p \in P} \ (\sum_{v \in V_p} c(\langle p, v \rangle)) = I_t(p)\}$$

  and for all $c \in dom(F_t)$, $F_t(c)$ is a finite bag and:

  $$\forall_{\langle\langle p,v\rangle,x\rangle \in F_t(c)} \ \ p \in O_t$$

Each place $p \in P$ has a set of allowed values (colours) attached to it and this means that a token residing in p must have a value v which is an element of this set, i.e. $v \in V_p$.

The function I specifies the bag of input places of each transition. If $t \in T$ and $p \in I_t$ then p is an input place of t with *multiplicity* $I_t(p)$. One can think of this multiplicity as the *weight* of the arc connecting the input place p and transition t. A transition $t \in T$ is enabled, if each of the input places contains at least the specified number of tokens, i.e. for all $p \in I_t$: there are at least $I_t(p)$ tokens in p. In the remainder of this monograph, we assume that for all $t \in T$: $I_t \neq [\ ]$, i.e. every transition has at least one input place. We also assume that TS is a subset of $\mathbb{R}^+ \cup \{0, \infty\}$, such that for all $x, y \in TS$: $x + y \in TS$.

The set of output places of each transition is specified by the function O. Note that O_t (for $t \in T$) is a set instead of a bag. The reason for this is the fact that the multiplicity of an output place is variable, i.e. the number of tokens produced for an output place may depend upon the values of the tokens consumed.

If $t \in T$ then F_t specifies the number of tokens produced (and their values and delays) given the values of the tokens consumed. The domain of F_t is the set of all possible bags of tokens consumed by t. Let $c \in dom(F_t)$ and $\langle\langle p, v\rangle, \langle x, y\rangle\rangle \in F_t(c)$. If transition t fires while consuming the tokens described by c, then t produces a token for place p with value v and a delay between x and y. Note that p has to be an output place of t.

To illustrate our rather formal definition of an ITCPN we give a small example:

$P = \{p_1, p_2\}$
$V_{p_1} = \mathbb{N}$ and $V_{p_2} = \{\text{‘}\texttt{signal}\text{’}\}$
$T = \{t_1, t_2\}$
$I = \{\langle t_1, [p_1]\rangle, \langle t_2, [p_2, p_2, p_2]\rangle\}$
$O = \{\langle t_1, \{p_1, p_2\}\rangle, \langle t_2, \emptyset\rangle\}$
For all $n \in \mathbb{N}$:
$F_{t_1}([\langle p_1, n\rangle]) = [\langle\langle p_1, n+1\rangle, \langle 1, 1\rangle\rangle], \quad \text{if } n < 10$
$F_{t_1}([\langle p_1, n\rangle]) = [\langle\langle p_1, 0\rangle, \langle 1, 1\rangle\rangle, \langle\langle p_2, \text{‘}\texttt{signal}\text{’}\rangle, \langle 0, 5\rangle\rangle], \quad \text{if } n \geq 10$
$F_{t_2}([\langle p_2, \text{‘}\texttt{signal}\text{’}\rangle, \langle p_2, \text{‘}\texttt{signal}\text{’}\rangle, \langle p_2, \text{‘}\texttt{signal}\text{’}\rangle]) = [\ ]$

Figure 2.8 shows the graphical representation of this ITCPN. The example describes a counter which produces a signal every 10 ‘ticks’ (with a delay between 0 and 5 ‘ticks’). There are two places and two transitions. Tokens in place p_1 have a numerical value (natural number) and tokens in place p_2 have a string value that equals ‘`signal`’. In this example, the input place of t_1 has multiplicity 1. The input place of t_2 has a multiplicity of 3, i.e. transition t_2 is enabled if there are at least three available tokens in place p_2. Function F_{t_1} describes the bag of tokens produced by the firing of t_1 given the value of the consumed token. Note that $dom(F_{t_1}) = \{[\langle p_1, n\rangle] \mid n \in \mathbb{N}\}$. If the value of the token in p_1 is smaller than 10, then t_1 produces one token for p_1 with a delay of precisely 1. Otherwise two tokens are produced, one for p_1 and one for p_2. The delay of the latter token is between 0

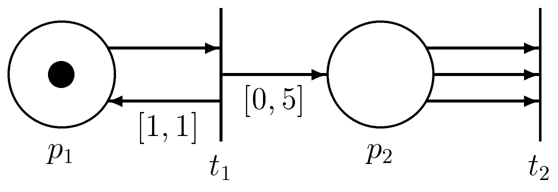


Figure 2.8: A graphical representation of an ITCPN

and 5. Transition t_2 only consumes tokens (in packets of three).

2.4.1 Semantics of an ITCPN

We describe the semantics of an ITCPN by a *transition system*, i.e. a pair $\langle S, R\rangle$ where S is the *state space* and $R \subseteq S \times S$ the *transition relation.*
In the transition system describing an ITCPN we attach a unique label (identification) to every token (in addition to the timestamp and value). Id is an infinite set of *token labels.* The *state space* of the transition system is:

$$S \;=\; Id \not\rightarrow (CT \times (TS \setminus \{\infty\})) \qquad (2.1)$$

So, in fact, a state $s \in S$ is a set of quartets representing: identity, position, value and timestamp, and the first one is unique. If $s \in S$ then $dom(s)$ is the set of token labels (identifications) corresponding to the tokens in the net. If $i \in dom(s)$ then $s(i)$ is a triplet representing the *position*, *value* and *timestamp* of the corresponding token. The timestamp of a token represents the time it becomes available, sometimes we refer to this time as the *arrival time* of a token. Note that we do not allow tokens to have a timestamp ∞, because there is no (intuitively) clear interpretation for this. For convenience we define a number of functions to refer to a specific aspect of a token.

Definition 11
For $q \in CT \times TS$ (or $q \in CT \times INT$) we define:

$$\begin{aligned} place(q) &= \pi_1(\pi_1(q)) \\ value(q) &= \pi_2(\pi_1(q)) \\ time(q) &= \pi_2(q) \end{aligned}$$

We call the firing of a transition an *event.* We define E to be the *event set*:

$$E \;=\; T \times S \times S \qquad (2.2)$$

An event changes a state into a new state, described by the transition relation. An event $e \in E$ is a triplet indicating the transition that fires ($\pi_1(e)$), the tokens consumed ($\pi_2(e)$) and the tokens produced ($\pi_3(e)$).

$AE(s) \subseteq E$ is the set of *allowed events* in state $s \in S$. An allowed event $e \in AE(s)$ satisfies a number of conditions. One of those conditions is: the delay of a produced token has to be sampled from the corresponding delay interval as specified by F. To sample a delay from the delay interval we introduce the concept of *specialization*. This concept is vital to a correct understanding of the semantics given in this section.

Definition 12 (Specialization)
For $s \in Id \not\rightarrow (CT \times TS)$ and $\overline{s} \in Id \not\rightarrow (CT \times INT)$: $s \lhd \overline{s}$ (s is a specialization of $\overline{s}$), if and only if, there exists a bijective function $f \in dom(s) \rightarrow dom(\overline{s})$ with: [2]

$$\begin{aligned} \forall_{i \in dom(s)} \quad & place(s(i)) = place(\overline{s}(f(i))) \;\wedge \\ & value(s(i)) = value(\overline{s}(f(i))) \;\wedge \\ & time(s(i)) \in time(\overline{s}(f(i))) \end{aligned}$$

If s is a specialization of $\overline{s}$ (i.e. $s \lhd \overline{s}$), then each token in s corresponds to precisely one token in $\overline{s}$ (and vice versa) such that they are in the same place, have the same value and the timestamp of the token in s is an element of the time interval of the token in $\overline{s}$. Figure 2.9 gives a graphical representation of the specialization concept, each token in s with identity i corresponds to a token in $\overline{s}$ with identity $f(i)$ such that $place(s(i)) = place(\overline{s}(f(i)))$, $value(s(i)) = value(\overline{s}(f(i)))$ and $time(s(i))$ is in the interval $time(\overline{s}(f(i)))$.

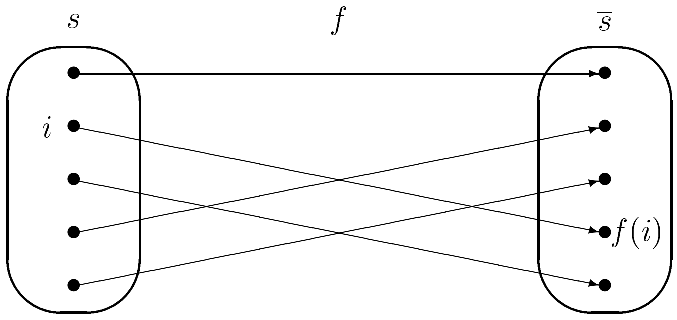


Figure 2.9: Specialization: $s \lhd \overline{s}$

To discard the timestamps of the tokens in a state, we define the function $untime \in S \rightarrow (Id \not\rightarrow CT)$. If $s \in S$ then:

$$untime(s) \quad = \quad \lambda_{i \in dom(s)} \; \langle place(s(i)), value(s(i)) \rangle \tag{2.3}$$

Now we can formalize $AE(s)$, the set of allowed events in state $s \in S$. An allowed

[2] If $x \in TS$ and $v \in INT$ then $x \in v \;\equiv\; \pi_1(v) \leq x \leq \pi_2(v) \;\wedge\; x < \infty$.

event $e \in AE(s)$ satisfies 5 conditions. The first condition is about the requirement that consumed tokens have to exist. The transition that fires consumes the correct number of tokens from the input places (condition (b)). Tokens are consumed in order of their timestamps (condition (c)). Produced tokens bear a unique label, condition (d) checks whether the label of a produced token does not exist already. Function F partially determines the bag of produced tokens. The delay of a produced token is sampled from the corresponding delay interval (condition (e)).

$$\begin{aligned}
AE(s) \;=\; & \{\langle t, q_{in}, q_{out}\rangle \in E \mid q_{in} \subseteq s \,\wedge && \text{(2.4a)}\\
& I_t = \lambda_{p\in P}\, \#\{i \in dom(q_{in}) \mid place(s(i)) = p\} \,\wedge && \text{(2.4b)}\\
& \forall_{i\in dom(q_{in})}\forall_{j\in dom(s)\setminus dom(q_{in})}\; place(s(i)) = place(s(j)) \Rightarrow && \\
& \qquad\qquad time(s(i)) \leq time(s(j)) \,\wedge && \text{(2.4c)}\\
& dom(q_{out}) \cap dom(s) = \emptyset \,\wedge && \text{(2.4d)}\\
& q_{out} \lhd \mathcal{BS}(F_t(\mathcal{SB}(untime(q_{in})))) \,\} && \text{(2.4e)}
\end{aligned}$$

For any event $\langle t, q_{in}, q_{out}\rangle \in AE(s)$, t is the transition which fires, q_{in} is the labelled bag of consumed tokens and q_{out} is the labelled bag of produced tokens. The tokens in q_{in} bear an 'absolute' timestamp. On the other hand, the timestamp of a token in q_{out} is 'relative', i.e. this timestamp represents the actual delay of the token.
Requirements (2.4a) and (2.4b) state that consumed tokens have to exist and that the number of tokens consumed from each place p is equal to the multiplicity of p. To satisfy the condition that timestamps have to be consumed in order of their timestamps, the timestamp of each consumed token has to be smaller or equal to the timestamp of any other token, which is not consumed by t and resides in the same place. This is stated by requirement (2.4c).
The last two requirements are about the tokens produced by the firing of t. First of all, the identity of each produced token is arbitrary as long as it is unique. This is stated by (2.4d) and the fact that q_{out} is a labelled bag. We use the specialization concept to state that the delays are sampled from the delay intervals of F_t, i.e. the actual delay of a produced token is between the upper and lower bound specified by F_t (see (2.4e)). Since the domain of F_t is a subset of $\mathbb{B}(CT)$, we have to use the function *untime* to delete the timestamps of the consumed tokens. The functions $\mathcal{BS}$ and $\mathcal{SB}$ are used to convert the bags into partial functions and vice versa. These functions are needed because the function F is defined in terms of bags and the transition system uses partial functions (i.e. labelled bags) to denote bags. Note that the identities of the produced tokens do not depend upon the definition of $\mathcal{BS}$, q_{out} is merely a specialization of $\mathcal{BS}(F_t(\mathcal{SB}(untime(q_{in}))))$.

In [58], Van Hee and Verkoulen describe a technique to to assign unique identifications to the produced tokens in a deterministic manner (based on the identifications of the consumed tokens). Although it is possible to use this technique for our model, we did not do this for reasons of convenience.

The timestamp of a token indicates the time it becomes available. The enabling time of a transition is the maximum timestamp of the tokens to be consumed. Because firing is an atomic action and transitions are eager to fire, we define the *event time* of an event $e \in E$ as follows:

$$et(e) \;=\; \max_{i \in dom(\pi_2(e))} \; time(\pi_2(e)(i)) \tag{2.5}$$

The *transition time* of a state $s \in S$ is the event time of the first event to occur, i.e. the minimum of the event times of the allowed events:

$$tt(s) \;=\; \min_{e \in AE(s)} \; et(e) \tag{2.6}$$

If there are two or more events with an event time equal to the transition time, then these events are in *conflict*. Conflicts are resolved non-deterministically.
Firing is an atomic action, thereby producing tokens with a timestamp of at least the firing time. The difference between the firing time and the timestamp of such a produced token is called the *firing delay*. In the transition system we have to add the firing time and the time delay. For this purpose we define the function *scale*. If $s \in S$ and $x \in TS$ then:

$$scale(s, x) \;=\; \lambda_{i \in dom(s)} \; \langle \langle place(s(i)), value(s(i)) \rangle, time(s(i)) + x \rangle \tag{2.7}$$

Finally, we define transition relation R. If $s_1, s_2 \in S$ then:

$$s_1 R s_2 \;\equiv\; \exists_{\substack{e \in AE(s_1) \\ et(e) = tt(s_1)}} \; s_2 = (s_1 \setminus \pi_2(e)) \;\cup\; scale(\pi_3(e), tt(s_1)) \tag{2.8}$$

If $s_1 R s_2$ then there is an event e transforming s_1 into s_2. This event consumes a number of tokens ($\pi_2(e)$) and produces zero or more tokens ($scale(\pi_3(e), tt(s_1))$). Note that the event time of the selected event is as small as possible, i.e. $et(e) = tt(s_1)$.

The complete transition system is summarized below:

The transition system
An ITCPN $= (P, V, T, I, O, F, TS)$ defines a *transition system* $\langle S, R \rangle$, with a state space S and a transition relation R:

- $S = Id \not\rightarrow (CT \times (TS \setminus \{\infty\}))$, the state space
- $E = T \times S \times S$, event set
- $untime(s) = \lambda_{i \in dom(s)} \; \langle place(s(i)), value(s(i)) \rangle$, delete timestamps from $s \in S$
- $AE(s) =$

$$\begin{aligned}
\{ \;\; & \langle t, q_{in}, q_{out} \rangle \;\in\; E \;\mid\; q_{in} \subseteq s \;\wedge \\
& I_t = \lambda_{p \in P} \; \#\{ i \in dom(q_{in}) \mid place(s(i)) = p \} \;\wedge \\
& \forall_{i \in dom(q_{in})} \forall_{j \in dom(s) \setminus dom(q_{in})} \; place(s(i)) = place(s(j)) \Rightarrow \\
& \qquad\qquad\qquad\qquad time(s(i)) \leq time(s(j)) \;\wedge \\
& dom(q_{out}) \cap dom(s) = \emptyset \;\wedge \\
& q_{out} \lhd \mathcal{BS}(F_t(\mathcal{SB}(untime(q_{in})))) \; \}
\end{aligned}$$

, set of allowed events in state $s \in S$

- $et(e) = \max_{i \in dom(\pi_2(e))} time(\pi_2(e)(i))$, event time of an event $e \in E$
- $tt(s) = \min_{e \in AE(s)} et(e)$, transition time of a state $s \in S$
- $scale(s, x) = \lambda_{i \in dom(s)} \langle\langle place(s(i)), value(s(i))\rangle, time(s(i)) + x\rangle$, scales the timestamps of the tokens in $s \in S$ with $x \in TS$
- Finally the transition relation R is defined by:

$$s_1 R s_2 \equiv \exists_{\substack{e \in AE(s_1) \\ et(e)=tt(s_1)}} \; s_2 = (s_1 \setminus \pi_2(e)) \cup scale(\pi_3(e), tt(s_1)) \quad \text{for any } s_1, s_2 \in S$$

We use a labelled bag to represent the state of an ITCPN. This is convenient, since it allows us to discriminate between tokens. However, the identification of a token is an arbitrary number and not very interesting from a modelling point of view. Moreover, definition 10 does not tell anything about identifications. Therefore, two states are called *equivalent* if and only if the corresponding bags are equal.

Definition 13
Let $s_1, s_2 \in S$ then s_1 and s_2 are equivalent ($s_1 \cong s_2$) if and only if $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$.

If two states, s_1 and s_2, are equivalent then s_1 can be transformed into s_2 (and vice versa) by relabelling the tokens in s_1 (see lemma 1).

2.4.2 Alternative firing rules

In section 1.3 we already mentioned that there are several ways to introduce quantitative time into Petri nets. One of the things one has to decide on is the *location* of the delay. We use a timing mechanism with time in tokens, firing is atomic and the transition determines the delay of a produced token. In this section we show that our style of semantics, that is a transition system with a state space $S = Id \to (CT \times (TS \setminus \{\infty\}))$, can be used to formalize alternative firing rules. We consider three alternative timing mechanisms: ‘place delays’, ‘enabling delays’ and ‘firing delays’. For simplicity we only consider deterministic delays, i.e. delays specified by a fixed value. Extensions to delays specified by an interval are straightforward.

Place delays

In [114], Sifakis proposes a model, called the Timed Place Transition Net model. In this model, time is associated with places, so that tokens arriving in a place are unavailable for a specified period. A token in a place may be in one of the following states: available or unavailable. For every unavailable token a time is given, this

time specifies when the token becomes available. The firing of a transition 'takes no time'. Besides Sifakis, there are other authors proposing place delays, consider for example Wong et al. [128].
It is easy to adapt the definition of the ITCPN model such that it is possible to specify place delays. Simply add the function PD to definition 10.

$$PD \in CT \rightarrow TS$$

This function specifies the length of time a token is unavailable. Note that this delay may depend upon the colour (value) of the token.
It is also easy to adapt the transition system such that it represents the formal semantics of the ITCPN model extended with place delays. Add the function $apt \in S \rightarrow S$ to the transition system. For $s \in S$:

$$\begin{aligned} apt(s) \;&= \lambda_{i \in dom(s)} \;\; \langle \langle place(s(i)), value(s(i)) \rangle, \\ &\qquad time(s(i)) + PD(\langle place(s(i)), value(s(i)) \rangle) \rangle \end{aligned}$$

And change formula (2.8) into:

$$s_1 R s_2 \;\; \equiv \;\; \exists_{\substack{e \in AE(s_1) \\ et(e)=tt(s_1)}} \; s_2 = (s_1 \setminus \pi_2(e)) \; \cup \; scale(apt(\pi_3(e)), tt(s_1)) \qquad (2.8')$$

The moment a token in place p and with value v becomes available, is delayed with $PD(\langle p, v \rangle)$ time units.
We just showed that it is easy to extend our ITCPN model with place delays, but is there really a need for this extension? We believe not, because our firing mechanism is a generalization of the firing mechanism using place delays. In the ITCPN model the transition determines the delay of a token, one can think of this delay as the time a token is unavailable. This is a generalization of place delays, since this delay may also depend on the transition producing the token.

Enabling delays

The majority of the timed Petri net models proposed in literature, associate time with the enabling time of a transition ([89], [16], [82], [41], [92], etc.). In these models a transition fires after a period of being continuously enabled. The firing of a transition takes no time. Suppose that the enabling time of each transition is given by:

$$ED \in T \rightarrow TS$$

Assuming that a transition t becomes enabled at time x and remains enabled until $x + ED_t$, then it will fire at time $x + ED_t$. If this transition becomes disabled before time $x + ED_t$, then there are two possible interpretations: (1) 'remember' the enabling time and start with this time when the transition becomes enabled again ('preemptive-resume'), (2) 'forget' about the enabling time, the enabling duration of a newly enabled transition is independent of any previous enabling ('preemptive-repeat'). Most authors use the later interpretation, so do we. This subject is discussed by Ajmone Marsan et al. in [81].

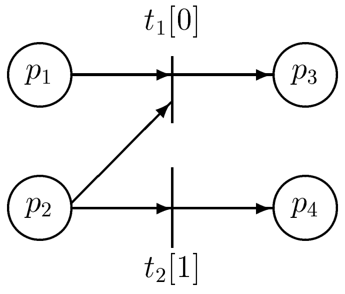


Figure 2.10: A 'timeout'

We also have to decide what to do with multiple enabledness of transitions. A transition t is twice enabled if every input place of t contains at least two times the required number of tokens (as specified by I_t). Different interpretations are possible now. For example, what to do if transition t becomes disabled once? It is difficult to decide on this. In our opinion the 'second' enabling is not affected by this disabling. It is easy to extend our ITCPN model with this kind of delay. Simply add the function $ED \in T \to TS$ to definition 10. The formal semantics of this extended model are given by a transition system similar to the transition system given in section 2.4.1. Simply change (2.5) into:

$$et(e) \;=\; \max_{i \in dom(\pi_2(e))} time(\pi_2(e)(i)) + ED_{\pi_1(e)} \tag{2.5$'$}$$

Associating time with the enabling of a transition is a very powerful concept. Enabling delays allow for the modelling of *priorities* and *timeouts*. With a priority we mean that if two transitions t_1 and t_2 are both enabled and share an input place, t_1 will fire for sure. Transition t_2 only fires if t_1 is not enabled. With a timeout we mean that a transition fires if a condition holds for a specified amount of time. Consider for example figure 2.10, $ED_{t_1} = 0$ and $ED_{t_2} = 1$. Suppose there is a token in place p_1 with timestamp x and there is a token in place p_2 with timestamp y. If $x < y + 1$ then t_1 will fire. If $x > y + 1$ then t_2 will fire at time $y + 1$. Transition t_2 represents a timeout, the token in place p_2 is 'lost' if it has been there for 1 time unit (i.e. consumed by t_2).

It is possible to model priorities and timeouts using our ITCPN. Consider for example figure 2.11, which shows an ITCPN corresponding to the net of figure 2.10. Both nets behave in a similar way. This example shows that modelling priorities and timeouts using an ITCPN is quite complex. There are, however, several reasons for the fact that we did not extend our ITCPN model with enabling delays. First of all, the concept of enabling delays allows for several interpretations (multiple enabledness, etc.). This makes it difficult to understand and to explain the model. Secondly, we believe that the number of timing mechanisms in the formal ITCPN model should be restricted to one. Multiple kinds of delays make the model more complex and difficult to use. Another reason for not choosing enabling delays is

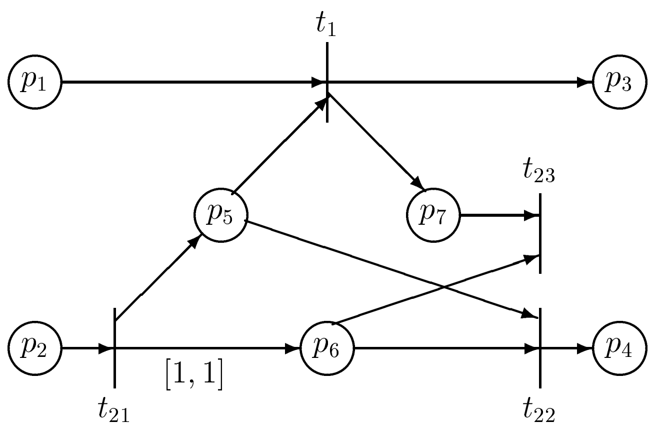


Figure 2.11: The 'timeout' modelled by an ITCPN

that we want to use the language ExSpect to specify ITCPNs and ExSpect does not support enabling delays.
Nevertheless, it is quite easy to add enabling delays to our ITCPN model. Furthermore, most of the concepts and techniques described in this monograph can be adapted to nets having enabling delays. This is demonstrated by the fact that the ITPN Analysis Tool (IAT) also supports the analysis of nets with enabling delays (see chapter 4).

Firing delays

The early timed Petri net models (e.g. [108], [107], [133]) associate a firing duration with each transition in the net. In these models the firing of a transition takes some time. Such a firing mechanism seems to be the most natural interpretation of time in transitions. Suppose that the firing duration of each transition is given by:

$$FD \in T \rightarrow TS$$

A transition with a positive firing duration is called a *timed transition*. Suppose a timed transition t becomes enabled at time x, at this moment the firing of t is initiated by removing tokens from the input places of t. The firing terminates at time $x + FD_t$, then the tokens are added to the output places of t. Note that firing is no longer atomic, therefore we call the firing of a timed transition a 'two-phase' firing. It is possible that a transition becomes enabled while it fires. Some authors allow multiple firings, i.e. a transition may be engaged in a number of firings at the same time. We do not allow multiple firings, i.e. a transition can not be enabled while it fires.

To give the formal semantics of this firing rule we have to change the transition system of section 2.4.1 radically. Therefore, we will give the semantics of this firing rule in terms of an ITCPN instead of a transition system, i.e. the meaning of this timing mechanism is given by a construction which replaces each transition by

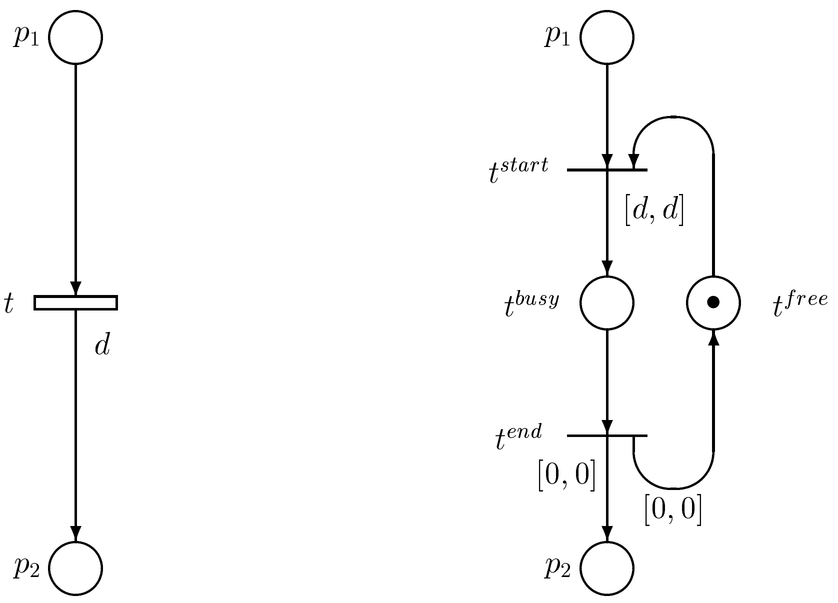


Figure 2.12: Construction of a timed transition (left) using two ITCPN transitions (right)

a small subnet. This construction is shown in figure 2.12. A timed transition t (represented by a small rectangle) is replaced by two (ITCPN) transitions t^{start} and t^{end}, and two places t^{free} and t^{busy}. Transition t^{start} is enabled if the input places of t contain enough tokens and place t^{free} contains a token. If t^{start} fires it adds one token to t^{busy} with a value representing the bag of tokens consumed from the input places and a delay $d = FD_t$. Transition t^{end} represents the termination of a firing.

More formally: suppose we have an ITCPN, say (P, V, T, I, O, F, TS), an initial state s and a function $FD \in T \to TS$ representing the firing delay of each transition. To construct the ITCPN, say $(\overline{P}, \overline{V}, \overline{T}, \overline{I}, \overline{O}, \overline{F}, \overline{TS})$, corresponding to (P, V, T, I, O, F, TS) with the transitions replaced by similar timed transitions, we select a timed transition $t \in T$ and define:

- $\overline{P} = P \cup \{t^{free}, t^{busy}\}$, such that $\{t^{free}, t^{busy}\} \cap P = \emptyset$
- $\overline{V}_{t^{free}} = \{\text{‘}\texttt{signal}\text{’}\}$, $\overline{V}_{t^{busy}} = \mathbb{B}(CT)$ and for all $p \in P$: $\overline{V}_p = V_p$
- $\overline{T} = (T \setminus \{t\}) \cup \{t^{start}, t^{end}\}$, such that $\{t^{start}, t^{end}\} \cap T = \emptyset$
- $\overline{I}_{t^{start}} = I_t \cup [t^{free}]$, $\overline{I}_{t^{end}} = [t^{busy}]$ and for all $\hat{t} \in (T \setminus \{t\})$: $\overline{I}_{\hat{t}} = I_{\hat{t}}$
- $\overline{O}_{t^{start}} = \{t^{busy}\}$, $\overline{O}_{t^{end}} = O_t \cup \{t^{free}\}$ and for all $\hat{t} \in (T \setminus \{t\})$: $\overline{O}_{\hat{t}} = O_{\hat{t}}$
- for all $\hat{t} \in (T \setminus \{t\})$: $\overline{F}_{\hat{t}} = F_{\hat{t}}$, and
  for all $c \in dom(\overline{F}_{t^{start}})$: $\overline{F}_{t^{start}}(c) = [\langle\langle t^{busy}, c \backslash [\langle t^{free}, \text{‘}\texttt{signal}\text{’}\rangle]\rangle, \langle FD_t, FD_t\rangle\rangle]$
  for all $c \in dom(\overline{F}_{t^{end}})$: $\overline{F}_{t^{end}}(c) = F_t(value(q)) \cup [\langle\langle t^{free}, \text{‘}\texttt{signal}\text{’}\rangle, \langle 0, 0\rangle\rangle]$,
  where q is the only element in the bag c ($\#c = 1$)

Repeat this until every timed transition is replaced by a subnet. The initial state of the constructed ITCPN, $(\overline{P}, \overline{V}, \overline{T}, \overline{I}, \overline{O}, \overline{F}, \overline{TS})$, is the initial state of (P, V, T, I, O, F, TS) with one token in each place of $\{t^{free} \mid t \in T\}$ with timestamp zero. Note that every transition $t \in T$ corresponds to precisely one unique place t^{free}. Similar statements hold for place t^{busy} and transitions t^{start} and t^{end}.
This construction gives our semantics of timed transitions. We will show that these semantics correspond to our conception of time in transitions.

Suppose we have an ITCPN, (P, V, T, I, O, F, TS), such that for all $t \in T$, $c \in dom(F_t)$ and $q \in F_t(c)$: $time(q) = \langle 0, 0 \rangle$, i.e. an ITCPN without delays. If we construct an ITCPN, $(\overline{P}, \overline{V}, \overline{T}, \overline{I}, \overline{O}, \overline{F}, \overline{TS})$, in the way described above, with the firing durations given by $FD \in T \to TS$, then the constructed net has a very specific structure. We will use this structure to prove a number of properties. In the rest of this section we assume that $\langle S, R \rangle$ is the transition system of the constructed ITCPN.

Lemma 5
For any $t \in T$ and $s_1, s_2 \in S$ such that $s_1 R s_2$, we have that if $X_t = \{t^{busy}, t^{free}\}$ then:

$$\#\{i \in dom(s_1) \mid place(s_1(i)) \in X_t\} = \#\{i \in dom(s_2) \mid place(s_2(i)) \in X_t\}$$

Proof.
Suppose $e \in AE(s_1)$ such that e is an event transforming s_1 into s_2. There are two possibilities: either there is a $t \in T$ such that $\pi_1(e) = t^{start}$ or there is a $t \in T$ such that $\pi_1(e) = t^{end}$. If $\pi_1(e) = t^{start}$, then a token is removed from place t^{free} and at the same time a token is added to place t^{busy}. Otherwise ($\pi_1(e) = t^{end}$) a token is removed from place t^{busy} and at the same time a token is added to place t^{free}. In both cases the total number of tokens in the places t^{busy} and t^{free} has not changed.
$\square$

The initial state of the constructed ITCPN is such that each place of $\{t^{free} \mid t \in T\}$ contains one token. This and lemma 5 imply that for any timed transition t there is a token in t^{busy} or there is a token in t^{free} but not in both. This property shows that a timed transition is either free or busy.

Lemma 6
Let $s_1, s_2 \in S$ such that $s_1 R s_2$ then:

$$\begin{aligned} & \forall_{i \in dom(s_1)}\ place(s_1(i)) \in P \ \Rightarrow\ time(s_1(i)) \leq tt(s_1) \\ \Rightarrow & \\ & \forall_{i \in dom(s_2)}\ place(s_2(i)) \in P \ \Rightarrow\ time(s_2(i)) \leq tt(s_2) \end{aligned}$$

Proof.
For every event $e \in AE(s_1)$, which transforms s_1 into s_2, there are two possibilities:

either $\pi_1(e) = t^{start}$ or $\pi_1(e) = t^{end}$ for some $t \in T$. If $\pi_1(e) = t^{start}$, then a token is removed from the places of $[t^{free}] \cup I_t$ and at the same time a token is added to place t^{busy}, i.e. t^{start} only consumes tokens from places in P. If $\pi_1(e) = t^{end}$, then a token is removed from place t^{busy} and at the same time tokens are added to some of the places in $\{t^{free}\} \cup O_t$. Every token added to a place in P has a delay of zero, because for all $t \in T$, $c \in dom(F_t)$ and $q \in F_t(c)$: $time(q) = \langle 0, 0 \rangle$. This and the monotonicity of time (see theorem 1 in section 2.5) imply that the timestamps of the tokens added to P are smaller than or equal to the new transition time ($tt(s_2)$).
$\square$

This lemma says that if initially each token in the places of P has a timestamp smaller than or equal to the transition time, then this remains so during the execution of any path. In other words, if all tokens in P are available in state s_1 then every state s_2 reachable by some sequence of events is such that each token in s_2 is available if it is located in a place of P. This lemma shows that the timestamps of the tokens in P do not affect the dynamic behaviour of the net, i.e. tokens in the places of P are always 'available'. Therefore, all timing aspects are restricted to the places added during the construction (in fact the places $\{t^{busy} \mid t \in T\}$).
Lemma 5 and lemma 6 illustrate the behaviour of the constructed net. We expatiated on this subject, because the construction of figure 2.12 is often used to model a resource with a finite capacity.

We have shown that our style of semantics can be used to formalize the meaning of various alternative firing mechanisms in a transparent and compact way. In each case the state space of the transition system is $S = Id \not\rightarrow (CT \times (TS \setminus \{\infty\}))$.
The majority of timed Petri net models proposed in literature represent a state as a pair $s = \langle m, d \rangle$ where m is the marking ($m \in \mathbb{B}(P)$) and d is the firing vector ($d \in T \not\rightarrow TS$ or $d \in T \not\rightarrow \mathbb{B}(TS)$). The firing vector represents the *residual* enabling (or firing) time of each enabled (or firing) transition in the net. If $t \in dom(d)$ then transition t completes (starts) its firing at time(s) d_t. When a transition t fires, both the marking and the firing vector have to be updated. Updating the firing vector involves a number of steps: (1) delete disabled transitions and t, (2) shift the residual enabling (firing) times and (3) add enabled transitions. The shift operation is necessary because these models use a relative time scale. For examples of timed Petri net models defined in such a manner, see [16], [17], [133], [81], [64] and [28].
Clearly, a transition system describing the semantics of a timed Petri net model using states of the form $\langle m, d \rangle$ is much more complex than the transition system given in section 2.4.1. Therefore, we associate time with tokens rather than places or transitions.

2.5 Some further concepts and properties

In this section we introduce some of the basic concepts and common terms normally used in Petri net theory. Because our ITCPN model is a timed high-level Petri

net model, some of these concepts have been extended. We also prove some of the behavioural properties of an ITCPN.
We use the following notations for the pre-set and post-set of a place p or a transition t:

$\bullet t = \{\hat{p} \in P \mid I_t(\hat{p}) > 0\}$	(the set of input places of t)
$t\bullet = O_t$	(the set of output places of t)
$\bullet p = \{\hat{t} \in T \mid p \in O_{\hat{t}}\}$	(the set of input transitions of p)
$p\bullet = \{\hat{t} \in T \mid I_{\hat{t}}(p) > 0\}$	(the set of output transitions of p)

An ITCPN is *conflict free*, if for each place p in the net the number of output transitions is smaller than or equal to 1, i.e. $\#(p\bullet) \leq 1$.
A place p without any input transition is called a *source place*, i.e. $\bullet p = \emptyset$.
A *sink place* is a place p without any output transition, i.e. $p\bullet = \emptyset$.

An ITCPN is called *ordinary*, if for each transition $t \in T$ of the net:

$$\begin{array}{ll} \forall_{p \in P} \;\; I_t(p) \leq 1 & \text{and} \\ \forall_{p \in O_t} \; \forall_{c \in dom(F_t)} \displaystyle\sum_{\substack{q \in F_t(c) \\ place(q)=p}} F_t(c)(q) \;= 1 & \end{array}$$

In other words, a net is ordinary if all 'multiplicities' (weights of input and output arcs) are equal to 1. Note that a transition in an ordinary net always produces exactly one token for each of its output places.

A *state machine* is an ordinary ITCPN such that each transition t has exactly one input place and one output place, i.e. $\forall_{t \in T} \; \#(\bullet t) = \#(t\bullet) = 1$.
A *marked graph* is an ordinary ITCPN such that each place p has one input transition and one output transition at the most, i.e. $\forall_{p \in P} \; \#(\bullet p) \leq 1 \;\wedge\; \#(p\bullet) \leq 1$. Some authors use the term *(timed) event graph* instead of marked graph.
A *free choice net* is an ordinary ITCPN such that for each place p with more than one output transition, this place is the only input place of each of these output transitions, i.e. $\forall_{p \in P} \; \#(p\bullet) \leq 1$ or $\bullet(p\bullet) = \{p\}$.[3]

A non-empty subset of places $X \subseteq P$ in an ITCPN, is called a *siphon* (also known as deadlock), if and only if, $\bullet X \subseteq X\bullet$, i.e. every transition having an output place in X has an input place in X. A siphon has the behaviour property that, if it is token free in some state s_1, then it remains token free in any state s_2 reachable from s_1. A non-empty subset of places $X \subseteq P$ in an ITCPN, is called a *trap* if $X\bullet \subseteq \bullet X$, i.e. every transition having an input place in X has an output place in X. If, in an ordinary net, a trap contains tokens, then in any successive state the trap contains tokens.

Sometimes we are only interested in the position of a token and not in its timestamp

[3]If $A \subseteq P$ or $A \subseteq T$, then $\bullet A = \cup_{a \in A} \; \bullet a$ and $A\bullet = \cup_{a \in A} \; a\bullet$.

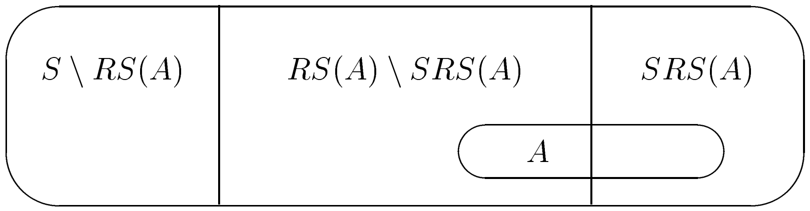


Figure 2.13: A partitioning of the state space S

or value. This leads to the definition of the *marking* of a state. A marking is denoted as a multiset of place indices. Function $M \in S \rightarrow \mathbb{B}(P)$ gives the marking of each state. If $s \in S$ then $M(s) = \lambda_{p \in P}\ \#\{i \in dom(s) \mid place(s(i)) = p\ \}$. The marking of a state represents the token distribution. For example, if $s \in S$ and $p \in P$ then $M(s)(p) = 3$ means that there are three tokens in place p.

In the remainder of this chapter we assume that $\langle S, R \rangle$ is the transition system describing the semantics of an ITCPN (P, V, T, I, O, F, TS). In section 2.3 we defined concepts such as reachability and process. These concepts are useful in the context of the transition system $\langle S, R \rangle$.
For an initial state $s \in S$, $R(s)$ is the set of states reachable by firing one transition in state s (see definition 4), i.e if $\hat{s} \in R(s)$ then there exists an allowed event e with $et(e) = tt(s)$ which transforms s into $\hat{s}$. If $A \subseteq S$ is a set of states, then $R(A)$ is the set of all states reachable by firing one transition in a state in A, i.e. $R(A) = \{\hat{s} \in S \mid \exists_{s \in A}\ sR\hat{s}\}$. $RS(A) = \cup_{n \in \mathbb{N}}\ R^n(A)$ is the set of all states reachable by firing an arbitrary number of transitions (when starting in a state in A).
The *process* of an ITCPN is described by the set of all possible *(execution) paths* (given a set of initial states A), i.e. $\Pi(A)$. A path $\sigma \in \Pi(A)$ is a sequence of states such that any successive pair belongs to the transition relation. The first state in a path is called the initial state and either the path is infinite or it ends in a terminal state (see definition 5). For all execution paths $\sigma \in \Pi(A)$ and $n \in \mathbb{N}$, $\sigma \restriction \{k \in \mathbb{N} \mid 0 \leq k < n\}$ is called a *firing sequence* (or *trace*). A firing sequence of length n describes $n - 1$ successive firings.

For a clear comprehension of the transition system describing the semantics of an ITCPN, it is useful to realize that there are three kinds of states. Suppose we have a set $A \subseteq S$ of possible initial states. In this case we partition the state space S into three classes, see figure 2.13. The first class, $SRS(A) = \{s \in S \mid \forall_{\sigma \in \Pi(A)}\ \exists_{i \in dom(\sigma)}\ \sigma_i = s\}$, consists of states visited by any execution path. The second class, $RS(A) \setminus SRS(A)$, represents the set of states which might be reached, i.e. these states are reachable, but they are not visited by every execution path stating in a state in A. The remaining states, $S \setminus RS(A)$, are the states not reachable when starting in a state in A.

For convenience we define the operation *place projection* ($|\!\!\upharpoonright$), returning the bag of timestamps of tokens in a certain place p given a state s.

Definition 14
For $s \in S$ and $p \in P$: $s|\!\!\upharpoonright p = \lambda_{x \in TS}\ \#\{i \in dom(s) \mid place(s(i)) = p \wedge time(s(i)) = x\}$

So, $\min(s|\!\!\upharpoonright p)$ is the smallest timestamp of the (non-empty) bag of tokens in place p.

Sometimes it is useful to know the *maximum* number of tokens in a place:

Definition 15
A place $p \in P$ is *K-bounded* in $s \in S$, if the number of tokens in p cannot exceed an integer K, i.e.

$$\forall_{\hat{s} \in RS(s)}\ \#(\hat{s}|\!\!\upharpoonright p) \leq K$$

A net is called K-bounded in $s \in S$ if all places are K-bounded in s. Nets that are 1-bounded are called *safe*. Places are often used to represent buffers. By verifying that the net is bounded or safe, it is guaranteed that there will be no overflow of any of the buffers, no matter what firing sequence is taken.

Definition 16
An ITCPN is called *conservative* with respect to a weighting function $W \in P \rightarrow \mathbb{R}$, if and only if, for all $s_1, s_2 \in S$ such that $s_1 R s_2$, the following relation holds:

$$\sum_{i \in dom(s_1)} W(place(s_1(i))) \ = \sum_{i \in dom(s_2)} W(place(s_2(i)))$$

All nets are conservative with respect to $W = \lambda_{p \in P}\ 0$. If the ITCPN is conservative with respect to $W = \lambda_{p \in P}\ 1$, then the ITCPN is said to be *strictly conservative*. In this case, the number of tokens does not change during any firing sequence. The concept of conservation is closely related to place invariants. In chapter 3 we will discuss how to generate invariants.

A path is a sequence of states. Consider the path $s_0, s_1, ..s_{i-1}, s_i, s_{i+1}, ...$ At time $tt(s_{i-1})$ an event occurred transforming state s_{i-1} into s_i. At time $tt(s_i)$ an event occurred transforming state s_i into s_{i+1}. Between $tt(s_{i-1})$ and $tt(s_i)$ the system was in state s_i. Since we are often interested in the state *at a certain moment in time*, we define H:

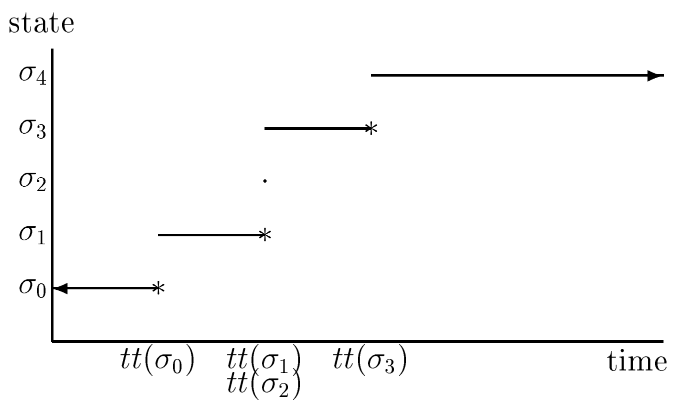


Figure 2.14: Relation between a path and the corresponding state function

Definition 17 (State function)
If $A \subseteq S$ and $\sigma \in \Pi(A)$ then $H(\sigma) \in TS \rightarrow S$ with:

$$\forall_{x \in TS} \; H(\sigma)(x) = \sigma_{\min\{i \in dom(\sigma) \;\mid\; x \leq tt(\sigma_i)\}}$$

is the *state function* of path σ.

The state function uses the following interpretation: at time x the ITCPN is in the first state having a transition time of at least x. Figure 2.14 shows the relation between a path and the corresponding state function. Note that at time $x = tt(\sigma_1) = tt(\sigma_2)$ the ITCPN is in state σ_1. However, several interpretations are possible, because firing is an atomic action.
When we defined the state space of the transition system describing the semantics of an ITCPN, we did not allow tokens to have a timestamp ∞. This allows us to formulate lemma 7.

Lemma 7
For a state $s \in S$: $s \in S^T$ if and only if $tt(s) = \infty$.

Proof.
The definition of S^T (the set of terminal states) shows that $s \in S^T$ implies that $tt(s) = \infty$. On the other hand, since every token has a timestamp smaller than ∞, the event time of any event is smaller than ∞. Hence, $tt(s) = \infty$ implies that there are no allowed events, i.e. $s \in S^T$.
$\square$

An important property of the ITCPN model is the monotonicity of time, i.e. time can only move forward. We use the following two lemmas to prove this.

Lemma 8
If $s_1, s_2 \in S$ and $s_1 \subseteq s_2$ then $tt(s_1) \geq tt(s_2)$.

Proof.
Observe that $s_1 \subseteq s_2$ means that state s_2 is state s_1 with zero or more additional tokens. First we show that:

$$\forall_{e_1 \in AE(s_1)} \exists_{e_2 \in AE(s_2)} \, et(e_2) \leq et(e_1)$$

Assume $e_1 = \langle t, q_{in}, q_{out} \rangle$ and $e_1 \in AE(s_1)$, then e_1 is such that the five conditions $(2.4a), .., (2.4e)$ on page 39 hold. Now we select an event $e_2 = \langle t, \hat{q}_{in}, \hat{q}_{out} \rangle$ such that $e_2 \in AE(s_2)$, this is possible because adding tokens cannot disable a transition. The fact that $e_2 \in AE(s_2)$ implies that condition $(2.4c)$ holds, therefore the tokens are selected from each input place of t in order of their timestamps. Event e_2 consumes tokens with timestamps smaller than or equal to the tokens in e_1, because s_2 is state s_1 with zero or more additional tokens. Therefore: $et(e_2) \leq et(e_1)$.
This implies that: $tt(s_1) = \min_{e_1 \in AE(s_1)} et(e_1) \geq \min_{e_2 \in AE(s_2)} et(e_2) \ = tt(s_2)$
□

Lemma 9
Let $s_1, s_2 \in S$ such that $dom(s_1) \cap dom(s_2) = \emptyset$. If for all $i \in dom(s_2)$: $time(s_2(i)) \geq tt(s_1)$, then $tt(s_1 \cup s_2) = tt(s_1)$.

Proof.
For any event $e \in AE(s_1 \cup s_2)$, either e consumes tokens from s_2 (i.e. $dom(\pi_2(e)) \cap dom(s_2) \neq \emptyset$) or not (i.e. $dom(\pi_2(e)) \cap dom(s_2) = \emptyset$). If e consumes tokens from s_2 then $et(e) \geq tt(s_1)$, because for all $i \in dom(s_2)$: $time(s_2(i)) \geq tt(s_1)$. Otherwise, $\pi_2(e) \subseteq s_1$. In this case $e \in AE(s_1)$ because the five conditions $(2.4a), .., (2.4e)$ on page 39 hold in state s_1 if they hold in state $s_1 \cup s_2$ ($\pi_2(e) \subseteq s_1$). This also implies that $et(e) \geq tt(s_1)$, i.e. $tt(s_1 \cup s_2) \geq tt(s_1)$. Lemma 8 tells us that $tt(s_1 \cup s_2) \leq tt(s_1)$, therefore $tt(s_1 \cup s_2) = tt(s_1)$.
□

Theorem 1 (Monotonicity)
Let $s \in S$, $\sigma \in \Pi(s)$ and $i, j \in dom(\sigma)$. If $i \leq j$ then $tt(\sigma_i) \leq tt(\sigma_j)$.

Proof.
First we prove that for all $s_1, s_2 \in S$ with $s_1 R s_2$: $tt(s_1) \leq tt(s_2)$. If $s_1 R s_2$ then there exists an event $e \in AE(s_1)$ such that $et(e) = tt(s_1)$ and $s_2 = (s_1 \setminus \pi_2(e)) \cup scale(\pi_3(e), tt(s_1))$. Using lemma 8 we see that deleting tokens ($\pi_2(e)$) does not decrease the transition time. The tokens of $scale(\pi_3(e), tt(s_1))$ have a timestamp of a least $tt(s_1)$. Using lemma 9 we deduce: $tt(s_1) \leq tt(s_2)$.
Note that $\sigma_i R^{j-i} \sigma_j$. We use induction to prove that for all $s_1, s_2 \in S$ and $n \in \mathbb{N}$: $s_1 R^n s_2 \Rightarrow tt(s_1) \leq tt(s_2)$. If $n = 0$ then $s_1 R^n s_2 \Rightarrow s_1 = s_2$ and $s_1 = s_2 \Rightarrow tt(s_1) \leq tt(s_2)$. Assume that for all $v \in S$: $s_1 R^{n-1} v \Rightarrow tt(s_1) \leq tt(v)$. Because $s_1 R^n s_2$ implies that there is a $v \in S$ with $s_1 R^{n-1} v$ and $v R s_2$, we deduce: $tt(s_1) \leq tt(v) \leq tt(s_2)$.

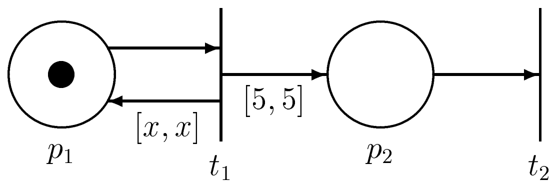


Figure 2.15: An ITCPN

□

This theorem shows that the transition times are ascending. Note that this does not imply that 'time moves forward' or 'time moves past a certain time'. Consider for example the ITCPN shown in figure 2.15. If the delay of the token produced for place p_1 is always 0 and initially there is a token in p_1 with timestamp 0, then t_1 will fire time after time but the transition time remains 0. In this case time does not move forward. Next, we define the ITCPN shown in figure 2.15 as follows:

$P = \{p_1, p_2\}$
$V_{p_1} = \mathbb{N}$ and $V_{p_2} = \{\text{`}\texttt{signal}\text{'}\}$
$T = \{t_1, t_2\}$
$I = \{\langle t_1, [p_1]\rangle, \langle t_2, [p_2]\rangle\}$
$O = \{\langle t_1, \{p_1, p_2\}\rangle, \langle t_2, \{\}\rangle\}$
For all $n \in \mathbb{N}$:
$F_{t_1}([\langle p_1, n\rangle]) = [\langle\langle p_1, n+1\rangle, \langle (1/2)^n, (1/2)^n\rangle\rangle, \langle\langle p_2, \text{`}\texttt{signal}\text{'}\rangle, \langle 5, 5\rangle\rangle]$
$F_{t_2}([\langle p_2, \text{`}\texttt{signal}\text{'}\rangle]) = [\ ]$

Let it be supposed that initially there is one token in p_1 with a value and timestamp equal to 0. Furthermore, assume that there are no tokens in p_2. Every time t_1 fires, the value of the token consumed from place p_1 is increased by 1 and restored in place p_1. The delay of the produced token is $(1/2)^n$, where n is the value of the token consumed. In this case time moves forward, but t_2 will never fire. The transition time of the k^{th} firing of t_1 is: $\sum_{0 \leq n < k-1} (1/2)^n$ and the enabling time of t_2 is 5. Consequently, transition t_2 will never fire, because $\lim_{k\to\infty} \sum_{0 \leq n < k-1} (1/2)^n = 2$.
This example shows that it is possible to specify an ITCPN with a dynamic behaviour which is in conflict with our intuition, i.e. time does not go by the way we think it should. This example demonstrates that we are in need of some *liveness concepts*. Many authors define liveness as follows: a Petri net is said to be *live* in a certain state s if, no matter what state has been reached from s, it is possible to fire any transition by progressing through some future firing sequence. Since we added time to our model, we are interested in liveness with respect to time. Therefore, we introduce a number of liveness concepts for interval timed coloured Petri nets.

Definition 18 (Liveness concepts)
For an initial state $s \in S$, an ITCPN is said to be:

dead,	if $\exists_{k \in \mathbb{N}}\ R^k(s) = \emptyset$
transient ,	if $\forall_{\sigma \in \Pi(s)}\ \forall_{i \in dom(\sigma)}\ \exists_{j \in dom(\sigma)}\ tt(\sigma_i) < tt(\sigma_j)$
livelock free,	if $\forall_{\sigma \in \Pi(s)}\ \forall_{i \in dom(\sigma)}\ \sigma_i \notin S^T \ \Rightarrow\ \exists_{j \in dom(\sigma)}\ tt(\sigma_i) < tt(\sigma_j)$
weakly progressive,	if $\forall_{x \in TS \setminus \{\infty\}}\ \exists_{\sigma \in \Pi(s)}\ \exists_{i \in dom(\sigma)}\ tt(\sigma_i) > x$
progressive ,	if $\forall_{x \in TS \setminus \{\infty\}}\ \forall_{\sigma \in \Pi(s)}\ \exists_{i \in dom(\sigma)}\ tt(\sigma_i) > x$

A net is dead in state s, if every path ends in a terminal state. Transience is a concept which characterizes nets where time never stops passing by, i.e. a net is transient in s if the time in the net continuously increases. Sometimes this concept is too strong. Thus, we relax the transience condition and define livelock free. A net is livelock free for an initial state, if the time in the net is increasing until a terminal state is encountered. A net is weakly progressive for an initial state, if there is no upper bound for the transition times, i.e. a net can reach an arbitrarily large time. A net is progressive, if an arbitrary time $x \in TS \setminus \{\infty\}$ can and will be reached.

The net shown in figure 2.15 (with delay $(1/2)^n$) is a non-progressive transient ITCPN in any state with a token in p_1. If there is no token in p_1, then the net is dead. Some of these liveness concepts are related. For example, if an ITCPN is dead in s, then it is also progressive in s. These relations are expressed in the following lemma:

Lemma 10
For an ITCPN and an initial state $s \in S$:

1. If the net is dead in s, then the net is progressive in s.
2. If the net is dead in s, then the net is not transient in s.
3. If the net is transient in s, then the net is livelock free in s.
4. If the net is progressive in s, then the net is weakly progressive in s.
5. If the net is progressive in s, then the net is livelock free in s.

Proof.
We only prove the first and the last property, the rest is easy to verify.

(1) Suppose the net is dead, then $\exists_{k \in \mathbb{N}}\ R^k(s) = \emptyset$. This implies that for any $\sigma \in \Pi(s)$ there exists a $k \in \mathbb{N} \setminus \{\infty\}$ such that $\#\sigma = k$. Note that σ_{k-1} is a terminal state, i.e. $\sigma_{k-1} \in S^T$. Recall that for every $\sigma_{k-1} \in S^T$: $tt(\sigma_{k-1}) = \infty$ (see lemma 7). Therefore, the net is progressive in s (see the definition of progressive).

(5) Suppose the ITCPN is progressive in s, i.e $\forall_{x \in TS \setminus \{\infty\}}\ \forall_{\sigma \in \Pi(s)}\ \exists_{i \in dom(\sigma)}\ tt(\sigma_i) > x$.

$$\begin{tikzcd}
 & \text{dead} \arrow[d] & \\
 & \text{progressive} \arrow[dl] \arrow[d] & \text{transient} \arrow[dl] \\
\text{weakly progressive} & \text{livelock free} &
\end{tikzcd}$$

Figure 2.16: Hierarchy of dynamic properties

Since $\{y \in TS \setminus \{\infty\} \mid i \in dom(\sigma) \ \wedge \ tt(\sigma_i) = y\} \subseteq TS \setminus \{\infty\}$, the progressiveness implies that:

$$\forall_{\sigma \in \Pi(s)} \ \forall_{x \in \{y \in TS \setminus \{\infty\} \ \mid \ i \in dom(\sigma) \ \wedge \ tt(\sigma_i) = y\}} \ \exists_{j \in dom(\sigma)} \ tt(\sigma_j) > x$$

Lemma 7 shows that: $\sigma_i \notin S^T$ if and only if $tt(\sigma_i) \neq \infty$. Hence:

$$\forall_{\sigma \in \Pi(s)} \ \forall_{\substack{i \in dom(\sigma) \\ \sigma_i \notin S^T}} \ \exists_{j \in dom(\sigma)} \ tt(\sigma_i) < tt(\sigma_j)$$

That is, the ITCPN is livelock free in s.
$\Box$

The relations between the liveness properties are shown in figure 2.16. In this monograph we often require a net to be progressive in the initial states. Therefore, we give sufficient conditions to guarantee that an ITCPN is progressive.

Lemma 11
Let an ITCPN be given with the additional properties: there is an $m \in \mathbb{N}$ and an $\epsilon \in TS$ such that $\epsilon > 0$ and:

$$\forall_{t \in T} \ \forall_{c \in dom(F_t)} \ (\#F_t(c) \leq m) \text{ and } (\forall_{b \in F_t(c)} \ \pi_1(time(b)) \geq \epsilon)$$

then the net is progressive for any initial state $s \in S$ having a finite number of tokens ($\exists_{n \in \mathbb{N}} \ \#s = n$).

Proof.
Let it be supposed that F satisfies the conditions mentioned and $s \in S$ such that $\#s = n$ ($n < \infty$). Now we have to prove that for any $\sigma \in \Pi(s)$:

$$\forall_{x \in TS \setminus \{\infty\}} \ \exists_{i \in dom(\sigma)} \ tt(\sigma_i) > x$$

We can prove this by showing that the following property holds for any $x \in TS \setminus \{\infty\}$:

$$\{i \in dom(\sigma) \mid tt(\sigma_i) \leq x\} \quad \text{is a } \textit{finite} \text{ set}$$

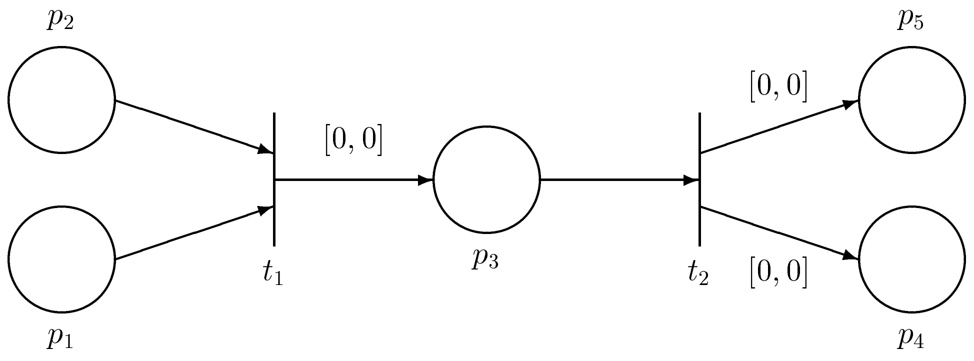


Figure 2.17: A (acyclic) progressive ITCPN

We prove this property for any $k \in \mathbb{N}$, $x = k\epsilon$, using induction.
For $k = 0$ the property holds because the number of tokens with a timestamp of 0 is finite ($\leq n$) and all produced tokens have a timestamp of at least ϵ, therefore the maximum number of firings with a transition time of 0 is n.
Assume, the property holds for $x = k\epsilon$, then we prove that it also holds for $x = (k+1)\epsilon$. The number of produced tokens with a timestamp in $(k\epsilon, (k+1)\epsilon]$ is finite, because (1) there are only finitely many firings possible with a transition time in $[0, k\epsilon]$ (induction hypothesis), (2) the number of tokens produced by every firing is finite (see conditions) and (3) we started with a finite number of tokens. Theorem 1 shows us that events with transition time later than $(k+1)\epsilon$ do not produce tokens for $(k\epsilon, (k+1)\epsilon]$. Events with a transition time in $(k\epsilon, (k+1)\epsilon]$ do not produce tokens with a timestamp in $(k\epsilon, (k+1)\epsilon]$ because of the minimal delay ϵ. Since the total number of tokens (produced and initially present) with a timestamp in $(k\epsilon, (k+1)\epsilon]$ is finite the number of firings with a transition time in $(k\epsilon, (k+1)\epsilon]$ is also finite. This implies that the number of firings in $(0, (k+1)\epsilon]$ is finite (induction hypothesis). Induction shows that this holds for any k and therefore for any $x \in TS \setminus \{\infty\}$ (use $k = \min\{l \in \mathbb{N} \mid l\epsilon \geq x\}$).
$\Box$

Lemma 11 gives us sufficient conditions to construct a progressive net. However, there are many progressive nets that do not satisfy the conditions stated in lemma 11. Consider for example the net shown in figure 2.17. This net contains delays equal to 0, nevertheless the net is progressive for any (finite) initial state s. To extend lemma 11 we define a *directed circuit* as follows:

Definition 19 (Circuit)
For an ITCPN, a (directed) circuit is a mapping $\rho \in \mathbb{N} \not\rightarrow T$ such that there exists an $n \in \mathbb{N}$ such that $dom(\rho) = \{k \in \mathbb{N} \mid k \leq n\}$, $\rho_n \bullet \cap \bullet \rho_0 \neq \emptyset$ and for all $i \in dom(\rho) \setminus \{0\}$: $\rho_{i-1} \bullet \cap \bullet \rho_i \neq \emptyset$.

Informally speaking: a circuit (or loop) is a sequence of interconnected transitions and places such that the last transition is connected to the first transition via some place. Note that the arcs connecting the places and transitions have to point in the proper direction. A net without circuits is called *acyclic*. It is easy to verify that an acyclic ITCPN is dead for any (finite) initial state:

Lemma 12
Let an *acyclic* ITCPN be given such that there exists an $m \in \mathbb{N}$ and:

$$\forall_{t \in T} \ \forall_{c \in dom(F_t)} \ \#F_t(c) \leq m$$

then the net is dead for any initial state $s \in S$ having a finite number of tokens ($\exists_{n \in \mathbb{N}} \ \#s = n$).

Proof.
Suppose, we have a net satisfying these conditions. For any token in state s the number of tokens produced directly and indirectly using this token is finite. If a token in a place p_1 is consumed during the firing of a transition, then this firing produces a finite number of direct successors ($\leq m$). Because the net is acyclic, these direct successors (i.e. tokens on the output places of the transition that fired) cannot be used to produce tokens for place p_1. Consider an arbitrary direct successor in some place p_2, this successor cannot be used to produce tokens for p_1 and p_2 (the net is acyclic), etc. Hence, the total number of successors of a token is smaller than $1 + m + m^2 + ..m^k$ with $k = \#P$. Initially, there are n tokens, therefore the maximum number of consecutive firings is $n(1+m+m^2+..m^k)$, i.e. the net is dead.
□

This lemma implies that an acyclic net is progressive (see lemma 10). The following theorem shows that if every circuit in a net contains a transition which produces tokens with a positive delay ($\geq \epsilon$), then the net is progressive (provided that the initial state has a finite number of tokens).

Theorem 2
Let an ITCPN be given with the additional properties:

$$\exists_{m \in \mathbb{N}} \ \forall_{t \in T} \ \forall_{c \in dom(F_t)} \ \#F_t(c) \leq m$$

and there is an $\epsilon > 0$ such that for every circuit ρ:

$$\exists_{i \in dom(\rho)} \ \forall_{c \in dom(F_{\rho_i})} \ \forall_{b \in F_{\rho_i}(c)} \ \pi_1(time(b)) \geq \epsilon$$

then the net is progressive for any initial state $s \in S$ having a finite number of tokens ($\exists_{n \in \mathbb{N}} \ \#s = n$).

Proof.
The proof of this theorem is similar to the proof of lemma 11. We prove progressiveness by showing that the following property holds for any $x \in TS \setminus \{\infty\}$:

$$\{i \in dom(\sigma) \mid tt(\sigma_i) \leq x\} \quad \text{is a } \textit{finite} \text{ set}$$

.

We prove this property for any $k \in \mathbb{N}$, $x = k\epsilon$, using induction. For $k = 0$ the property holds because the initial number of tokens with a timestamp of 0 is finite ($\leq n$) and the number of tokens produced with a timestamp 0 is finite. The number of tokens produced with a timestamp 0 is finite because we can omit at least one transition in every circuit ρ, without effecting the behaviour at time 0. Note that in every circuit ρ there is a transition ρ_i with $i \in dom(\rho)$ such that $\forall_{c \in dom(F_{\rho_i})} \forall_{b \in F_{\rho_i}(c)} \pi_1(time(b)) \geq \epsilon$, this means that ρ_i produces tokens with a timestamp of at least ϵ. If we (temporarily) remove these transitions we have an acyclic net. lemma 12 tells us an acyclic net is dead. Hence, the number of firings with a transition time of 0 is finite.
Assume that the property holds for $x = k\epsilon$, then we have to prove that it also holds for $x = (k+1)\epsilon$. The number of produced tokens with a timestamp in $(k\epsilon, (k+1)\epsilon]$ is finite, because (1) there are only finitely many firings possible with a transition time in $[0, k\epsilon]$ (induction hypothesis), (2) the number of tokens produced by every firing is finite (see conditions), (3) we started with a finite number of tokens and (4) every circuit contains a transition with only positive delays. Theorem 1 shows us that events with transition time later than $(k+1)\epsilon$ do not produce tokens for $(k\epsilon, (k+1)\epsilon]$. Events with a transition time in $(k\epsilon, (k+1)\epsilon]$ produce a finite number of tokens with a timestamp in $(k\epsilon, (k+1)\epsilon]$, because of we can disregard at least one transition in every circuit (delay $\geq \epsilon$), i.e. for the firings in $(k\epsilon, (k+1)\epsilon]$ it suffices to consider an acyclic net. Lemma 12 tells us an acyclic net is dead. Hence, the total number of produced tokens with a timestamp in $(k\epsilon, (k+1)\epsilon]$ is finite. Since the total number of tokens with a timestamp in $(k\epsilon, (k+1)\epsilon]$ is finite, the number of firings with a transition time in $(k\epsilon, (k+1)\epsilon]$ is also finite. This implies that the number of firings in $(0, (k+1)\epsilon]$ is finite (induction hypothesis). Induction shows that this holds for any k and therefore for any x.
$\square$

Theorem 2 enables us to recognise the progressiveness of many nets by observing the definition of the net only, i.e. we can prove that an ITCPN is progressive without considering the set of reachable states or possible firing sequences. Figure 2.18 shows a net having a circuit and a delay 'zero'. Yet, we can prove that this net is progressive (for any finite initial state), by applying theorem 2.

2.6 Interesting performance measures

It is useful to show that an ITCPN satisfies certain properties, such as progressiveness and boundedness. However, we are also in need of concepts to calculate the *performance* of the system modelled by an ITCPN. With performance we mean characteristics, such as: response times, occupation rates, transfer rates, throughput times, failure rates, etc.

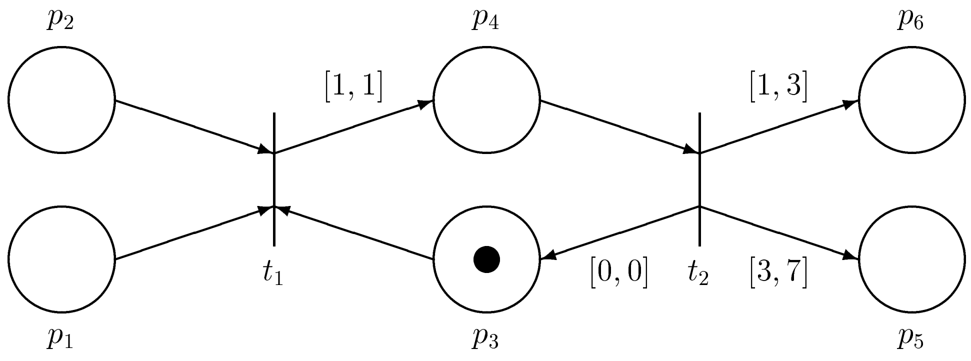


Figure 2.18: A progressive ITCPN

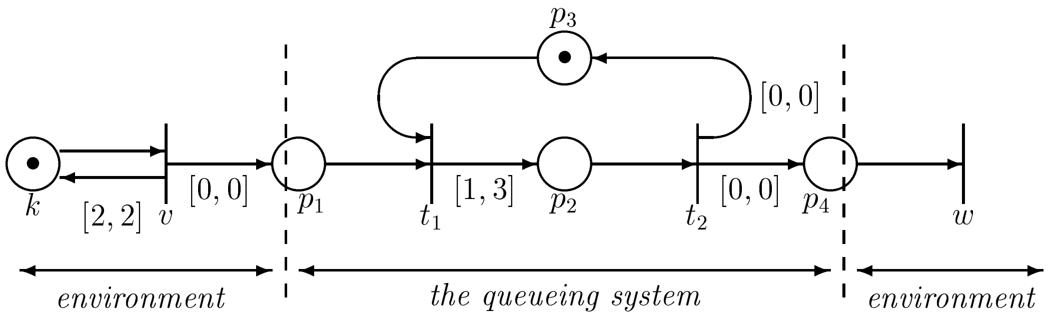


Figure 2.19: A queueing system whose environment is modelled explicitly

When analysing the performance of a system, there are three important aspects: (1) the behaviour of the system, (2) the initial state of the system and (3) the behaviour of the environment of the system. Clearly, performance measures such as occupation rates and response times also depend upon the initial state of the system (e.g. the initial number of capacity resources) and the environment of the system (e.g. the number of requests per hour).

The fact that the performance of a system depends on the behaviour of environment, stimulated many authors working on (timed) Petri nets to model the environment of the system explicitly. Consider for example the single server queue shown in figure 2.19. Tokens in place p_1 represent arriving customers (e.g. jobs). Every job requires some service (service time between 1 and 3). There is only one server (e.g. a machine) modelled by a token in place p_2 or p_3 (but not in both). Jobs leave the system via place p_4.

If we want to analyse the performance of this net (e.g. throughput), then we may decide to model the environment explicitly. To model the arrival of jobs we add an extra place (k) and a transition (v). If the initial state is such that there is one token

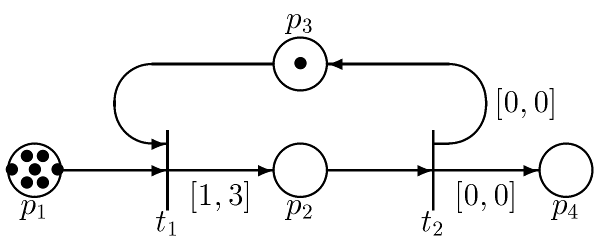


Figure 2.20: A queueing system whose environment is simulated by the initial state

in place k, then the interarrival time of jobs is equal to 2. If we want to analyse the system under various circumstances, we have to adapt the net definition.

The ITCPN model allows for an alternative approach. This approach uses the initial state to represent the behaviour of the environment of the system. Now it is possible to analyse the system under various circumstances without having to change the net description. Figure 2.20 shows the single server queue modelled by an ITCPN with an initial state which also specifies the behaviour of the environment. Initially p_1 contains tokens with timestamps describing the time of their arrival. In this approach, the net is considered to be a function or algorithm that can be applied to some initial state, i.e. given an initial state the net 'calculates' the dynamic behaviour of the system.

Note that it is not possible to use this approach to model environments which 'interact' with the system, i.e. an environment which gives feedback. However, the ITCPN model also allows for the explicit modelling of complex reactive environments, which cannot be modelled using the initial state.

In many cases it is very convenient to simulate the environment by choosing a suitable initial state, because we often want to analyse a number of alternatives under various circumstances. The latter approach prevents us from having to change the net description every time we vary the load of the system. In a way, this approach looks upon the net as a 'black box' which responds to inputs generated by the environment. Another advantage of this approach is that it allows for a stepwise analysis of large nets. Consider for example figure 2.21, where the rectangles A, B, C and D represent subnets. In this example, we are able to analyse subnet A in isolation, because A is not influenced by the rest of the net. A thorough analysis of subnet A gives us all possible 'inputs' for subnets B and C. If we have analysed B and C, then we can analyse D. Now we are able to tell something about the 'overall' performance of the system.

There are two reasons why most authors model the environment explicitly. The first reason for this is that they use models with time in transitions or time in places instead of time in tokens. Consequently, they are unable to express events

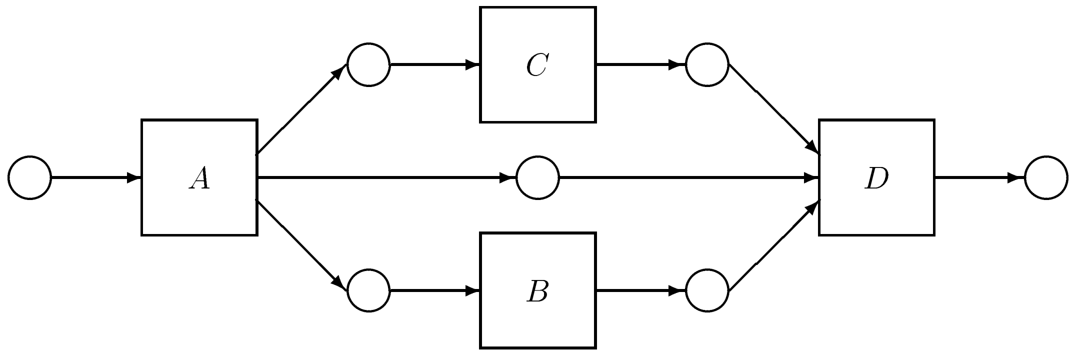


Figure 2.21: Stepwise analysis of a large net

and conditions in the future using the initial state only. Consider for example the queueing system shown in figure 2.20, to specify the arrival of an extra job at time x, they need to add an extra transition. The second reason is that they are interested in the steady-state behaviour of a system. A steady-state functioning of the net is only possible if the environment has some 'regular' behaviour. In this case, it suffices to model the environment by a simple subnet.

What are the typical performance measures defined in Petri net literature?
People working on deterministic timed Petri nets are mainly interested in the minimal cycle time of a periodically operated Petri net. The cycle time is the time it takes to complete a firing sequence leading to a state having a marking equal to the initial state. See [62], [107], [28] and [113] for further information.
Researchers using stochastic timed Petri nets are interested in the steady-state distribution, i.e. the probability of being in a specific marking. It is possible to derive several interesting performance measures from such a steady-state distribution, see for example Ajmone Marsan et al. [83] or [80].

Many of systems we are interested in, are not stable, i.e. we also consider processes having an initial transient period and processes which never stabilize. Consider for example a production unit, at the beginning and ending of a working-day there are all kinds of disturbances and the load of the production unit may vary during the day. The fact that we use interval timing and our interest in the analysis of non-stationary processes forces us to develop new performance measures. These are defined in the remainder of this section.

If one models systems where time aspects are important, one is often interested in characteristics, such as throughput times and response times. This is the reason we developed the measures *earliest* and *latest first arrival time* for a place in the net.
The earliest (latest) first arrival time of a place p is the largest (smallest) lower (upper) bound for the timestamp of the 'first' token in place p (given some initial state).

Definition 20 ($\mathcal{EAT}, \mathcal{LAT}$)
Given an ITCPN, a state $s \in S$ and a place $p \in P$ we define:

$$\begin{aligned} \mathcal{EAT}(s,p) &= \min_{\sigma \in \Pi(s)} \min_{i \in dom(\sigma)} \min(\sigma_i \upharpoonright p) \\ \mathcal{LAT}(s,p) &= \max_{\sigma \in \Pi(s)} \min_{i \in dom(\sigma)} \min(\sigma_i \upharpoonright p) \end{aligned}$$

for the *earliest first arrival time* and the *latest first arrival time* respectively.

To clarify these concepts we give a small example. Let the ITCPN shown in figure 2.20 be defined by:

$P = \{p_1, p_2, p_3, p_4\}$
$V_{p_1} = V_{p_4} = \{\text{`}\texttt{job}\text{'}\}$
$V_{p_2} = \{\text{`}\texttt{busy}\text{'}\}$
$V_{p_3} = \{\text{`}\texttt{free}\text{'}\}$
$T = \{t_1, t_2\}$
$I = \{\langle t_1, [p_1, p_3]\rangle, \langle t_2, [p_2]\rangle\}$
$O = \{\langle t_1, \{p_2\}\rangle, \langle t_2, \{p_3, p_4\}\rangle\}$
$F_{t_1}([\langle p_1, \text{`}\texttt{job}\text{'}\rangle, \langle p_3, \text{`}\texttt{free}\text{'}\rangle]) = [\langle\langle p_2, \text{`}\texttt{busy}\text{'}\rangle, \langle 1, 3\rangle\rangle]$
$F_{t_2}([\langle p_2, \text{`}\texttt{busy}\text{'}\rangle]) = [\langle\langle p_3, \text{`}\texttt{free}\text{'}\rangle, \langle 0, 0\rangle\rangle, \langle\langle p_4, \text{`}\texttt{job}\text{'}\rangle, \langle 0, 0\rangle\rangle]$

Let it be supposed that we have an initial state s with one token in place p_1 and one token in place p_3, and both tokens have a timestamp 0. It is easy to see that: $\mathcal{EAT}(s,p_1) = \mathcal{LAT}(s,p_1) = 0$ and $\mathcal{EAT}(s,p_3) = \mathcal{LAT}(s,p_3) = 0$. In this case, t_1 fires at time 0 followed by a firing of t_2 at some time between 1 and 3. This implies that: $\mathcal{EAT}(s,p_2) = 1$, $\mathcal{LAT}(s,p_2) = 3$, $\mathcal{EAT}(s,p_4) = 1$ and $\mathcal{LAT}(s,p_4) = 3$.
Note that $\mathcal{EAT}(s,p)$ and $\mathcal{LAT}(s,p)$ are only defined for the *first* token to 'arrive' in p. However, it is possible to generalize these concepts for a set of initial states $A \subseteq S$ and n tokens:

Definition 21 ($\mathcal{EAT}_n, \mathcal{LAT}_n$)
For an ITCPN, a set of states $A \subseteq S$, a place $p \in P$ and $n \in \mathbb{N} \setminus \{0\}$ we define:

$$\begin{aligned} \mathcal{EAT}_n(A,p) &= \min_{\sigma \in \Pi(A)} \min_{i \in dom(\sigma)} \text{bmin}_n(\sigma_i \upharpoonright p) \\ \mathcal{LAT}_n(A,p) &= \max_{\sigma \in \Pi(A)} \min_{i \in dom(\sigma)} \text{bmin}_n(\sigma_i \upharpoonright p) \end{aligned}$$

where $\text{bmin}_n b = \min_{\hat{b} \subseteq b \ \wedge \ \#\hat{b} = n}(\max \hat{b})$.

If a bag $b \in \mathbb{B}(TS)$ contains at least n elements, then $\text{bmin}_n b$ is the n^{th} timestamp in the bag (selected in ascending order), otherwise $\text{bmin}_n b$ is infinite.
If $\mathcal{EAT}_n(A,p) = x$, then x is the smallest value such that there exists a path starting in a state $s \in A$ that visits a state with at least n tokens in p each with a timestamp

less or equal to x. If $\mathcal{LAT}_n(A,p) = x$, then x is the largest value such that there exists a path such that all the states visited by this path do not have n tokens in p each with a timestamp smaller than x. Note that $\mathcal{EAT}_n(\{s\},p) = \mathcal{EAT}(s,p)$ and $\mathcal{LAT}_n(\{s\},p) = \mathcal{LAT}(s,p)$.
If p is a *sink* place (i.e. $p\bullet = \emptyset$), then $\mathcal{EAT}_n(A,p)$ can be interpreted as a lower bound for the arrival time of the n^{th} token, that is *earliest n^{th} arrival time*. In this case, $\mathcal{LAT}_n(A,p)$ can be interpreted as the *latest n^{th} arrival time*.
Again, we use the net shown in figure 2.20 to illustrate these performance measures. Suppose we have an initial state $s = \{\langle -1, \langle\langle p_3, \text{`}\texttt{free}\text{'}\rangle, 0\rangle\rangle\} \cup \{\langle i, \langle\langle p_1, \text{`}\texttt{job}\text{'}\rangle, 2i\rangle\rangle \mid i \in \mathbb{N}\}$, i.e. a state with one token in p_3 (timestamp 0) and an infinite number of tokens in p_1 (timestamp $2i$). Note that the interarrival time between two jobs is 2 time units. If $n \in \mathbb{N} \setminus \{0\}$ then $\mathcal{EAT}_n(s,p_1) = \mathcal{LAT}_n(s,p_1) = 2(n-1)$, $\mathcal{EAT}_n(s,p_4) = 2n-1$ and $\mathcal{LAT}_n(s,p_4) = 3n$. The throughput time of the n^{th} job, i.e. waiting time and service time, is between $\mathcal{EAT}_n(s,p_4) - \mathcal{LAT}_n(s,p_1) = 1$ and $\mathcal{LAT}_n(s,p_4) - \mathcal{EAT}_n(s,p_1) = n+2$.

The following lemma tells us that it is also possible to define the earliest and latest n^{th} arrival time (i.e. $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$) recursively.

Lemma 13

$$\mathcal{EAT}_n(s,p) = \text{bmin}_n(s \upharpoonleft p) \text{ min } \text{min}_{\hat{s}\in R(s)} \mathcal{EAT}_n(\hat{s},p)$$
$$\mathcal{LAT}_n(s,p) = \text{bmin}_n(s \upharpoonleft p) \text{ min } \text{max}_{\hat{s}\in R(s)} \mathcal{LAT}_n(\hat{s},p)$$

Proof.
We derive the first equation in a number of steps:

$$\begin{aligned}
& \text{bmin}_n(s \upharpoonleft p) \text{ min } \text{min}_{\hat{s}\in R(s)} \mathcal{EAT}_n(\hat{s},p) \\
= \quad & \lbrace\!\lbrace \text{ definition of } \mathcal{EAT}_n \rbrace\!\rbrace \\
& \text{bmin}_n(s \upharpoonleft p) \text{ min } \text{min}_{\hat{s}\in R(s)}(\text{min}_{\hat{\sigma}\in\Pi(\hat{s})} \text{min}_{i \in dom(\hat{\sigma})} \text{ bmin}_n(\hat{\sigma}_i \upharpoonleft p)) \\
= \quad & \lbrace\!\lbrace\ \hat{s} \in R(s) \ \wedge \ \hat{\sigma} \in \Pi(\hat{s}) \Leftrightarrow \sigma \in \Pi(s) \text{ (where } \sigma = \text{``}s\hat{\sigma}\text{''}) \rbrace\!\rbrace \\
& \text{bmin}_n(s \upharpoonleft p) \text{ min } (\text{min}_{\sigma\in\Pi(s)} \text{min}_{i \in dom(\sigma)\setminus\{0\}} \text{ bmin}_n(\sigma_i \upharpoonleft p)) \\
= \quad & \lbrace\!\lbrace\ \Pi(s) \neq \emptyset \rbrace\!\rbrace \\
& \text{min}_{\sigma\in\Pi(s)}(\text{bmin}_n(s \upharpoonleft p) \text{ min } \text{min}_{i \in dom(\sigma)\setminus\{0\}} \text{ bmin}_n(\sigma_i \upharpoonleft p)) \\
= \quad & \lbrace\!\lbrace\ \sigma_0 = s \rbrace\!\rbrace \\
& \text{min}_{\sigma\in\Pi(s)} \text{min}_{i \in dom(\sigma)} \text{ bmin}_n(\sigma_i \upharpoonleft p) \\
= \quad & \lbrace\!\lbrace \text{ definition of } \mathcal{EAT}_n \rbrace\!\rbrace \\
& \mathcal{EAT}_n(s,p)
\end{aligned}$$

Note that we use brackets ($\lbrace\!\lbrace\ \rbrace\!\rbrace$) to delimit comments. There is an analogous proof for the latest n^{th} arrival time.
□

We use $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$ to measure characteristics, such as throughput times and response times. Another interesting characteristic of a system is the utilization of a resource, consider for example the occupation rate of a machine or the stock level in a distribution centre. These performance measures are closely related to the number of tokens in a certain place during the execution of the net. Because our model is non-deterministic, we start with the definition of the average number of tokens in a place *given an execution path.*

Definition 22 (U)
If $s \in S$, $\sigma \in \Pi(s)$, $p \in P$, $t \in TS$ and $t > 0$ then:

$$U(\sigma, p, t) = \frac{1}{t} \int_0^t M(H(\sigma)(x))(p)\ \lambda(d\,x)$$

is the *average number of tokens* in p during $[0, t]$, where λ is the Lebesgue measure.

Now we are able to define a lower and an upper bound for the *occupation rate* of a place.

Definition 23 ($\mathcal{LOR}, \mathcal{HOR}$)
If $s \in S$, $p \in P$, $t \in TS$ and $t > 0$ then we define:

$$\begin{aligned} \mathcal{LOR}(s,p,t) &= \min\nolimits_{\sigma \in \Pi(s)} U(\sigma, p, t) \\ \mathcal{HOR}(s,p,t) &= \max\nolimits_{\sigma \in \Pi(s)} U(\sigma, p, t) \end{aligned}$$

for the *lowest occupation rate* and *highest occupation rate* respectively.

Given an initial state s the average number of tokens in p during $[0, t]$ is between $\mathcal{LOR}(s,p,t)$ and $\mathcal{HOR}(s,p,t)$. This allows us to analyse logistical concepts, such as machine utilization and stock levels.
For the net shown in figure 2.20, $n \in \mathbb{N}\backslash\{0\}$ and an initial state $s = \{\langle -1, \langle\langle p_3, \text{‘}\texttt{free}\text{’}\rangle, 0\rangle\rangle\} \cup \{\langle i, \langle\langle p_1, \text{‘}\texttt{job}\text{’}\rangle, 2i\rangle\rangle \mid i \in \mathbb{N}\}$: $\mathcal{LOR}(s, p_1, n) = \infty$, $\mathcal{HOR}(s, p_1, n) = \infty$, $\mathcal{HOR}(s, p_2, n) = 1$ and $\mathcal{LOR}(s, p_2, 2n) = 0.5$. These last two figures tell us that the occupation rate of the server is between 0.5 and 1, because there is one token in place p_2 if and only if the server is busy.

The simplicity of the queueing system example allowed us to calculate performance measures, such as $\mathcal{LOR}, \mathcal{HOR}, \mathcal{EAT}_n$ and $\mathcal{LAT}_n$ manually. For large and complex nets it is not possible to do this by hand. Therefore, we are in need of efficient and powerful tools for the automatical calculation of these measures. This is the reason we developed a number of analysis methods, which are presented in the following chapter. Based on these analysis methods we also developed a software tool, called *IAT*, to analyse interval timed coloured Petri nets. This tool is described in chapter 4.

2.7 Conclusion

In this chapter we have defined the ITCPN model. Compared to conventional timed Petri net models, there are three notable differences:
The first difference with conventional timed Petri net models is the fact that we have a high-level model, i.e. tokens are coloured. Many authors have extended the basic Petri net model with *coloured* or *typed tokens* ([46], [70], [132], [53]). In these models tokens have a value, often referred to as colour. There are several reasons for such an extension. One of these reasons is the fact that (uncoloured) Petri nets tend to become too large to handle. Another reason is the fact that tokens often represent objects or resources in the modelled system. As such, these objects may have attributes, which are not easily represented by a simple Petri net token. A 'coloured' Petri net model allows the modeller to make much more succinct and manageable descriptions. Although several high-level Petri net models have been proposed in literature, only a few of these models also incorporate time.
The second difference with conventional timed Petri nets is the fact that time is in tokens and each token bears a unique label, this we adopted from Van Hee et al. [58]. As a result, our ITCPN model has transparent semantics (considering the fact that we have a coloured Petri net model with interval timing) and a very compact state representation ($S = Id \not\rightarrow (CT \times (TS \setminus \{\infty\}))$). We have shown that the complete formal semantics of our model fits on one page, see section 2.4.1. In our model, firing is atomic and the transition which fires determines the delays of the tokens produced. We also investigated alternative firing rules, e.g. place delays and enabling delays. We have demonstrated that our timing mechanism is suitable for the modelling of discrete dynamic systems. Nevertheless, it is quite easy to add other timing mechanisms to the ITCPN model (see section 2.4.2).
The third difference is the fact that the firing delay is non-deterministic *and* non-stochastic. In our model we use intervals to describe time delays. Specifying the delay by means of an interval rather than a deterministic value or a stochastic variable, allows for the representation of time constraints. This is very important when modelling time-critical systems. Examples of such systems are real-time (computer) systems and just-in-time manufacturing systems.

To our knowledge, only one other model has been presented in literature which also uses delays specified by an interval. This model was presented by Merlin in [89] and [90]. In this model the enabling time of a transition is specified by a minimal and a maximal time. Another difference with our model is the fact that Merlin's model is not a high-level Petri net model because of the absence of typed (coloured) tokens. Compared to our model, Merlin's model has a rather complex formal semantics, which was presented in [16] by Berthomieu and Diaz. This is caused by a redundant state space (marking and enabled transitions are represented separately) and the fact that they use a relative time scale and allow for multiple enabledness of transitions (see section 2.4.2).

We use a transition system to describe the semantics of the ITCPN. This transition system has been used to define a number of concepts in a compact and elegant manner.

The fact that we use interval timing and our interest in processes without a ‘steady-state’ behaviour forced us to develop a number of new performance measures. In section 2.6, we have defined the measures: $\mathcal{EAT}_n, \mathcal{LAT}_n, \mathcal{LOR}$ and $\mathcal{HOR}$. $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$ are used to calculate upper and lower bounds for characteristics, such as throughput times and response times. $\mathcal{LOR}$ and $\mathcal{HOR}$ are used to estimate measures, such as occupation rates, stock levels and average queue lengths.
In the next chapter, we will discuss methods to calculate these performance measures and to verify several behavioural properties. To prove the correctness of these analysis methods, we will use the preliminaries given in section 2.2 and section 2.3.

Chapter 3

Analysis of time in nets

3.1 Introduction

In this chapter we present an approach to verify certain properties and to calculate bounds for all sorts of performance measures. This approach is based on a number of new analysis methods, three of which are presented in this chapter. These methods have in common that they utilize the interval timing aspect of our ITCPN model.

In chapter 1 we expressed our interest in discrete dynamic systems, i.e. systems characterized by the words: discrete, dynamic and distributed. Petri nets extended with time and colour are appropriate for the modelling of these systems, in particular logistic systems (this will be demonstrated in chapter 5). Therefore, we have developed the ITCPN model defined in the previous chapter.
Modelling a complex discrete dynamic system in terms of an ITCPN is useful for a number of reasons. First of all, the ITCPN model serves as an aid to thought, since model building forces us to organize, evaluate and examine the validity of our thoughts. Since we are interested in distributed systems, the graphical nature of Petri nets agrees with the applications we have in mind. Secondly, we can formalize certain properties of the system. In section 2.5 we stated a number of interesting properties, for example the absence of traps and siphons (deadlocks), progressiveness and boundedness. Thirdly, we can use an ITCPN to analyse the performance of the system. In section 2.6 we defined a number of interesting performance measures.
In most cases, performance analysis and the verification of certain properties are the main goals of model building. For this reason we have developed a number of analysis methods based on our ITCPN model.

In section 1.4 we already mentioned other analysis techniques applicable to Petri nets. Only a few of these techniques have been developed (or extended) for the analysis of *timed and coloured* Petri nets. Existing techniques which can be used to analyse the dynamic behaviour of such nets, may be subdivided into three classes:

- simulation
- reachability analysis

- Markovian analysis

Simulation is a technique to analyse a system by conducting controlled experiments (see Shannon [112]). These experiments are used to verify the correctness of the model and to predict the behaviour of the system under consideration. Because simulation does not require difficult mathematical techniques, it is easy to understand for people with a non-technical background. Simulation is also a very powerful analysis technique, since it does not set additional restraints. However, sometimes simulation is expensive in terms of the computer time necessary to obtain reliable results. Another drawback is the fact that (in general) it is not possible to use simulation to *prove* that the system has the desired set of properties (at least not the properties we are interested in, see section 2.5 and section 2.6). Nevertheless, extensive simulation can be used to test certain assumptions and to predict performance measures (and their accuracy).
Recent developments in computer technology stimulate the use of simulation for the analysis of timed coloured Petri nets. The increased processing power allows for the simulation of large nets. Modern graphical screens are fast and have a high resolution. Therefore, it is possible to visualize a simulation graphically (i.e. animation).

Reachability analysis is a technique which constructs a reachability graph, sometimes referred to as reachability tree or occurrence graph (cf. Jensen [71], Peterson [100], Murata [93]). Such a reachability graph contains a node for each possible state and an arc for each possible state change. Reachability analysis is a very powerful method in the sense that it can be used to prove all kinds of properties. Another advantage is the fact that it does not set additional restraints.
Obviously, the reachability graph needed to prove these properties may, even for small nets, become very large (and often infinite). If we want to inspect the reachability graph by means of a computer, we have to solve this problem. This is the reason several authors developed reduction techniques (Hubner et al. [67] and Valmari [120]). Unfortunately, it is not known how to apply these techniques to timed coloured Petri nets.

For timed coloured Petri nets with certain types of stochastic delays it is possible to translate the net into a *continuous time Markov chain*. This Markov chain can be used to calculate performance measures like the average number of tokens in a place and the average firing rate of a transition.
If all the delays are sampled from a negative exponential probability distribution, then it is easy to translate the timed coloured Petri net into a continuous time Markov chain. Several authors attempted to increase the modelling power by allowing other kinds of delays, for example mixed deterministic and negative exponential distributed delays, and phase-distributed delays (see Ajmone Marsan et al. [80]). Nearly all stochastic Petri net models (and related analysis techniques) do not allow for coloured tokens, because the increased modelling power is offset by computational difficulties. This is the reason stochastic high-level Petri nets are often used

in a simulation context only. Nevertheless, a number of stochastic high-level net models have been proposed in literature (Lin and Marinescu [76], Zenie [131] and Dutheillet and Haddad [38]).

Besides the aforementioned techniques to analyse the behaviour of timed coloured Petri nets, there are several analysis techniques for Petri nets without 'colour' or explicit 'time'.

An interesting way to analyse a coloured Petri net is to calculate (or verify) *place and transition invariants* (P and T-invariants). Place and transition invariants can be used to prove properties of the modelled system. A mapping $W \in CT \rightarrow \mathbb{Z}$ is a place invariant, if for all $s_1, s_2 \in S$ such that $s_1 R s_2$, the following relation holds:[1]

$$\sum_{i \in dom(s_1)} W(\langle place(s_1(i)), value(s_1(i))\rangle) \;=\; \sum_{i \in dom(s_2)} W(\langle place(s_2(i)), value(s_2(i))\rangle)$$

Intuitively, a place invariant assigns a weight to each token such that the weighted sum of all tokens in the net remains constant during the execution of any firing sequence. By calculating these place invariants we find a set of equations which characterizes all reachable states. Transition invariants are the duals of place invariants and the main objective of calculating transition invariants is to find firing sequences with no 'effects'.
Note that we can calculate these invariants for *timed* coloured Petri nets (e.g. an ITCPN). However, in this case, we do not really use the timing information. Therefore, in general, these invariants do not characterize the dynamic behaviour of the system. On the other hand, they can be used to verify properties which are timing independent.
For more information about the calculation of invariants in a coloured Petri net, see Jensen et al. [71], [72] and [69].

In our ITCPN model, a delay is described by an interval rather than a fixed value or some delay distribution. On the one hand, interval delays allow for the modelling of variable delays, on the other hand, it is not necessary to determine some artificial delay distribution (as opposed to stochastic delays). Instead, we have to specify bounds. These bounds are used to specify and to verify time constraints. This is very important when modelling time-critical systems, i.e. *real-time* systems with 'hard' deadlines. These deadlines have to be met for a safe operation of the system. An acceptable behaviour of the system depends not only on the logical correctness of the results, but also on the time at which the results are produced. Therefore, we are interested in techniques to verify these deadlines and to calculate upper and lower bounds for all sorts of performance criteria.

[1]This definition of a place invariant is the straightforward extension of place invariants for uncoloured nets. Other authors (e.g. Jensen [70]) use a slightly more complicated definition, where the weight function maps token colours into multisets over a common colour set A (instead of integers), i.e. $W \in CT \rightarrow \mathbb{B}(A)$.

To our knowledge, for Petri nets with interval timing, only one analysis method has been proposed which really uses this timing behaviour. This method was presented by Berthomieu, Diaz and Menasche in [17] and [16], and uses Merlin's timed Petri nets ([89]) to describe the system. The method generates a reachability graph where nodes represent state classes instead of states. This approach is more or less related to one of the analysis methods presented in this chapter.

Since this method is based on Merlin's timed Petri net model, there are some serious drawbacks. First of all, the model does not allow for coloured tokens. This implies that it is difficult to make manageable models for large and complex systems. Secondly, this analysis method can only be used for nets with the environment modelled explicitly, because time is associated with transitions rather than tokens (see section 2.6). Thirdly, they use a relative time scale, which prohibits the calculation of the performance measures defined in section 2.6, e.g. the upper and lower bound for the arrival time of the n^{th} token in a place p (i.e. $\mathcal{EAT}_n(s,p)$ and $\mathcal{LAT}_n(s,p)$, see definition 21). Furthermore, as a result of the fact that this method uses a relative time scale, it is not possible to verify liveness properties such as progressiveness and transience.

To meet these problems, the author of this monograph has developed four new analysis methods, all based on the ITCPN model. This chapter deals with three of these methods.

The most powerful method we have developed is the *Modified Transition System Reduction Technique* (MTSRT), described in section 3.3. The MTSRT method can be applied to an arbitrary ITCPN. This method generates a *reduced reachability graph*. In an ordinary reachability graph, a node corresponds to a state. To calculate such an ordinary reachability graph, we start with an initial state. For every state s, we obtain 'new states'. These are the states reachable by firing a transition in state s. New states are connected to s by an arc. Repeating this process results in a graph representation of the reachable states. Even for simple examples, these graphs tend to be very large (in general infinite). The MTSRT method proposes a number of reductions, resulting in a reduced reachability graph. In such a graph a node corresponds to a set of places, called a *state class*, instead of a single state. To generate a graph representation of these state classes, we use a modified transition system, where a time *interval* is associated with a token rather than a timestamp.

The other three methods can only be applied to a restricted set of interval timed Petri nets.

The *Persistent Net Reduction Technique* (PNRT) can only be applied to persistent nets. In section 3.4 we will investigate the behaviour of such nets. The PNRT method uses the special structure of a persistent net to create an even further reduced reachability graph. This method is quite efficient and calculates $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$.

The *Arrival Times in Conflict Free Nets* (ATCFN) method can be applied to conflict

free nets, i.e. nets where the number of output arcs of each place is smaller than or equal to 1. This method produces upper and lower bounds for the arrival time of the first token in a place using a polynomial-time algorithm. This method will be presented in section 3.2.
We have developed one method, called the *Steady State Performance Analysis Technique* (SSPAT), to analyse periodically operated Petri nets. The SSPAT method calculates upper and lower bounds for the cycle time of a net. This is a generalization of the technique described by Ramamoorthy and Ho in [107]. The SSPAT method has been presented in Van der Aalst [2].

To keep the size of the reduced reachability graph generated by the MTSRT (or PNRT) method manageable, it may be necessary to simplify the ITCPN by means of *refinements* in combination with *uncolouring*. The basic idea behind this approach is to ignore certain aspects of complex token colours. This idea is also the starting point of the concept of *colour set restrictions* mentioned by Jensen in [71] and the concept of *projections* introduced by Genrich in [43]. However, our approach can be applied to *timed* coloured Petri nets. In section 3.5 we will show that refining or uncolouring does not affect the validity of the analysis results. We use an example to illustrate this approach (section 3.6).

3.2 Method ATCFN

The first analysis method we present, called *Arrival Times in Conflict Free Nets* (ATCFN), calculates bounds for the arrival time of the first token in a place, i.e. given an initial state s and a place p, this method calculates $\mathcal{EAT}(s,p)$ and $\mathcal{LAT}(s,p)$. Unfortunately, this method can only be applied to conflict free progressive ITCPNs, where all input arcs have multiplicity 1. However, in section 3.2.1, we will show that this is not a serious restriction in the field of project engineering. Furthermore, if we consider an ITCPN that does not satisfy these restrictions (conflict free, progressive, multiplicity 1), then the results produced by this algorithm can be interpreted as lower bounds for $\mathcal{EAT}(s,p)$ and $\mathcal{LAT}(s,p)$.
There is some similarity with 'the Dijkstra algorithm' to calculate the shortest path ([37]) and the methods to calculate the earliest event times in an activity network, e.g. CPM and PERT (see Price [105], Lock [77] and Whitehouse [125]). It is in fact an extension to the situation with two node types: transitions and places.

In order to describe the algorithm, we have to quantify the relation between a transition and a place.

Definition 24 (D^{min}, D^{max})
Given an ITCPN, a transition t and a place p:

$$\begin{array}{rcl} D^{min}(t,p) & = & \min_{c \in dom(F_t)} \min\{\pi_1(time(q)) \mid q \in F_t(c) \ \wedge \ place(q) = p\} \\ D^{max}(t,p) & = & \max_{c \in dom(F_t)} \min\{\pi_2(time(q)) \mid q \in F_t(c) \ \wedge \ place(q) = p\} \end{array}$$

$D^{min}(t,p)$ ($D^{max}(t,p)$) is the minimal (maximal) difference in time between the firing of t and the 'arrival' of the first token in p produced by this firing. Recall, we use the term arrival time to refer to the time a token becomes available, i.e. its timestamp. If p is not an output place of t, then $D^{min}(t,p) = \infty$ and $D^{max}(t,p) = \infty$. An interpretation of $D^{min}(t,p)$ ($D^{max}(t,p)$) is the minimal (maximal) 'time distance' between a transition t and a place p. Note that this distance does not depend on the values of the consumed tokens. Since the ITCPN model associates delays with produced tokens rather than consumed tokens, the distance between a place and a transition is zero.

First, we consider the algorithm to calculate $\mathcal{EAT}$ given an initial state $s \in S$. In this algorithm we assign a label to each place in the net. There are two kinds of labels: *permanent* and *tentative* labels. A label has a (time) value indicating the earliest arrival time of the first token in the corresponding place.
We represent the set of places bearing a permanent label by X_p and the set of places bearing a tentative label by X_t. The value of each label is given by $d^{min} \in P \to TS$. For a place p with a permanent label, $d^{min}(p)$ is the earliest arrival time of a token in p, i.e. if $p \in X_p$, then $d^{min}(p) = \mathcal{EAT}(s,p)$. If $p \in X_t$, then $d^{min}(p)$ is the earliest arrival time found so far.
Initially, each place bears a tentative label. In the algorithm the set X_p is extended successively.

Algorithm for the calculation of $\mathcal{EAT}(s,p)$

step 1 Assign a tentative label to each place in the net ($X_t = P$, $X_p = \emptyset$). For each place p, the (time) value is set to the smallest timestamp of the tokens initially present in p. If, initially, there are no tokens in p, then the value of the label is set at ∞. In other words: $d^{min}(p) = \min(s \upharpoonright p)$.

step 2 If there are no places with a tentative label and a finite (time) value, then terminate. Otherwise, select a place p with a tentative label and the smallest value (i.e. $p \in X_t$ and $d^{min}(p) = \min\{d^{min}(l) \mid l \in X_t\}$). Declare the label of p to be permanent instead of tentative.

step 3 Consider all transitions t satisfying the following conditions: p is an input place of t and all input places bear a permanent label ($t \in p\bullet$ and $\bullet t \subseteq X_p$).

For every such t, consider all output places k that bear a tentative label ($k \in (t\bullet) \cap X_t$). If the value of the label attached to k is greater than $d^{min}(p) + D^{min}(t,k)$, then change the value of the label attached to k to $d^{min}(p) + D^{min}(t,k)$.

If all relevant transitions t with the corresponding output places k have been considered, then go to step 2.

Alternatively, we can give a more compact description of the algorithm using 'pseudo-code', see figure 3.1. There is a similar algorithm to calculate $\mathcal{LAT}$: D^{min} and d^{min}

```
input ITCPN,$s$

$X_t := P$;
$X_p := \emptyset$;
for $p \in P$ do $d^{min}(p) = \min(s \upharpoonright p)$ end;

while  $\min\{d^{min}(l) \mid l \in X_t\} < \infty$
       do
       select $p \in X_t$ with $d^{min}(p) = \min\{d^{min}(l) \mid l \in X_t\}$;
       $X_t := X_t \setminus \{p\}$;
       $X_p := X_p \cup \{p\}$;
       for  $t \in \{v \in p\bullet \mid \bullet v \subseteq X_p\}$
            do
            for $= k \in (t\bullet) \cap X_t$
                do
                $d^{min}(k) := d^{min}(k) \min (d^{min}(p) + D^{min}(t,k))$;
                end;
            end;
       end;

output $X_t, X_p, d^{min}$
```

Figure 3.1: A description of the algorithm ATCFN in pseudo-code

are replaced by D^{max} and d^{max}. The following theorem shows us that these algorithms calculate $\mathcal{EAT}$ and $\mathcal{LAT}$ for a restricted class of nets.

Theorem 3
Let $s \in S$ be the initial state of an ITCPN that satisfies three conditions: (1) the ITCPN is conflict free, (2) the ITCPN is progressive in s and (3) all input arcs have multiplicity 1. For any place $p \in P$, we have:

$$\begin{aligned} d^{min}(p) &= \mathcal{EAT}(s,p) \\ d^{max}(p) &= \mathcal{LAT}(s,p) \end{aligned}$$

Proof.
We prove this theorem by showing that there exists an invariant and a termination argument. The outer loop in the algorithm satisfies four invariant relations (see figure 3.1):

$Q1(X_t, X_p, d^{min})$**:** $\quad X_p \cup X_t = P$ and $X_p \cap X_t = \emptyset$

$Q2(X_t, X_p, d^{min})$**:** $\quad \forall_{k \in X_p} \forall_{l \in X_t}\ d^{min}(k) \leq d^{min}(l)$

$Q3(X_t, X_p, d^{min})$**:** $\quad \forall_{k \in X_p}\ d^{min}(k) = \mathcal{EAT}(s,k)$

$$Q4(X_t, X_p, d^{min}): \quad \forall_{k \in X_t} \; d^{min}(k) = (\min s \upharpoonleft k) \min \\ (\min_{\substack{v \in T \\ \bullet v \subseteq X_p}} \max_{l \in \bullet v} d^{min}(l) + D^{min}(v,k))$$

Initially, $X_t = P$, $X_p = \emptyset$ and for all $l \in P$: $d^{min}(l) = \min(s \upharpoonleft l)$. It is easy to show that each of the invariant relations holds after initialization.

Suppose the invariant relations hold just before an element p is transferred from X_t to X_p, i.e. $Q1 = Q1(X_t, X_p, d^{min})$, $Q2 = Q2(X_t, X_p, d^{min})$, $Q3 = Q3(X_t, X_p, d^{min})$ and $Q4 = Q4(X_t, X_p, d^{min})$ hold, $p \in X_t$ and $d^{min}(p) = \min\{d^{min}(l) \mid l \in X_t\}$.
If $X_t' = X_t \setminus \{p\}$, $X_p' = X_p \cup \{p\}$ and $d^{min\prime}$ is the updated mapping (see step 3), then we have to prove that $Q1' = Q1(X_t', X_p', d^{min\prime})$, $Q2' = Q2(X_t', X_p', d^{min\prime})$, $Q3' = Q3(X_t', X_p', d^{min\prime})$ and $Q4' = Q4(X_t', X_p', d^{min\prime})$ hold.
In is easy to see that $Q1'$ holds. Invariant $Q2'$ also holds, because p is a minimal element of X_t.
$Q3'$ holds, because $d^{min}(p) = \mathcal{EAT}(s,p)$, this follows from $Q2$,$Q3$ and $Q4$. To prove this, observe the subexpression $(\min_{\substack{v \in T \\ \bullet v \subseteq X_p}} \max_{l \in \bullet v} d^{min}(l) + D^{min}(v,k))$ of Q4. Since, all input places l are permanent, we have $d^{min}(l) = \mathcal{EAT}(s,l)$ (use $Q3$). It is sufficient to consider transitions with permanent input places only, because all transitions having a tentative input place do not fire before $d^{min}(p)$ (use $Q2$). Furthermore, a transition v will fire at its enabling time, because the net is conflict free and progressive. Therefore, the value of this subexpression is equal to the smallest possible timestamp of a token in p produced by any transition.
If the smallest possible timestamp of a token in p was not produced by a transition, then it was initially there, i.e. $\mathcal{EAT}(s,p) = \min\ (s \upharpoonleft p)$. Using $Q4$ this implies that $d^{min}(p) = \mathcal{EAT}(s,p)$ (i.e. $Q3'$ holds).
Invariant $Q4'$ is violated by the transfer of p from X_t to X_p. This is repaired by the two inner 'for loops', see figure 3.1.

The algorithm terminates, because the number of elements in X_t is decreasing. The remaining places in X_t are not reachable and got the value ∞ initially.
Note that we need the three conditions to prove this theorem, i.e. if we drop one of the conditions, then it is not guaranteed that $\max_{p \in \bullet v} \mathcal{EAT}_n(s,p)$ is the earliest possible firing time of transition v.
An analogous proof holds for the upper bound of the first arrival.
□

This theorem tells us that the algorithm can be used to calculate $\mathcal{EAT}$ and $\mathcal{LAT}$ for a restricted class of nets. A serious restriction is the fact that conflicts between transitions are not allowed. If there are conflicts in the net, for example to model shared resources, the algorithm can give incorrect results. However, sometimes it is possible to model certain kinds of parallelism and synchronization without having conflicts.
The condition that the net has to be progressive in s is not very restrictive. In section 2.5 we gave sufficient conditions to guarantee progressiveness.

If the ITCPN does not satisfy the conditions mentioned in theorem 3, then $d^{min}(p) \leq \mathcal{EAT}(s,p)$ and $d^{max}(p) \leq \mathcal{LAT}(s,p)$ (for an arbitrary ITCPN, any place p and any initial state $s \in S$), i.e. the algorithm produces lower bounds for $\mathcal{EAT}$ and $\mathcal{LAT}$. For an arbitrary net, the first token in place p does not arrive before $d^{min}(p)$ and it is possible to construct a firing sequence where the first token does not arrive before $d^{max}(p)$.

Theorem 4
For an arbitrary ITCPN, any place $p \in P$ and any initial state $s \in S$, we have:

$$\begin{array}{rcl} d^{min}(p) & \leq & \mathcal{EAT}(s,p) \\ d^{max}(p) & \leq & \mathcal{LAT}(s,p) \end{array}$$

Proof.
The proof of this theorem is analogous to the proof of theorem 3. Replace invariant relation $Q3(X_t, X_p, d^{min})$ with:

$$\overline{Q3}(X_t, X_p, d^{min}) \equiv \forall_{k \in X_p} \; d^{min}(k) \leq \mathcal{EAT}(s,k)$$

Suppose the invariant relations hold just before an element p is transferred from X_t to X_p.
First, we prove the invariance of $Q1$, $Q2$ and $Q4$, this can be done in the same way as in the proof of theorem 3. The proof of the invariance of $\overline{Q3}$ is slightly different. If we drop one of the conditions stated in theorem 3, then $\max_{p \in \bullet v} \; \mathcal{EAT}_n(s,p)$ is merely a lower bound for earliest possible firing time of transition v. If the net contains conflicts, then v may become disabled. If the net is not progressive in s, then it is not guaranteed that time progresses past the enabling time of v. Finally, multiple input arcs may delay the enabling time of a transition. Consequently, omitting one (or more) of the conditions of theorem 3 results in the calculation of lower bounds for $\mathcal{EAT}(s,p)$ and $\mathcal{LAT}(s,p)$.
$\Box$

Now let us consider the complexity of the algorithm ATCFN. Clearly, the computing time depends upon the number of places $n = \#P$, the number of transitions $m = \#T$ and the number of tokens in the initial state $l = \#s$.
The worst-case (time) complexity of the algorithm presented in this section is $\mathcal{O}(l + mn^2)$ and it requires $\mathcal{O}(n)$ storage space.[2] However, the algorithm ATCFN is usually a lot faster. If the number of output transitions of each place is smaller than some constant c and the number of tokens in the initial state is rather small, then the worst-case (time) complexity of the algorithm is quadratic in the number of places (i.e. $\mathcal{O}(n^2)$). Since these assumptions are quite reasonable, the computational cost of our algorithm is comparable to the computational cost required by

[2]Let $f(n)$ and $g(n)$ be two functions of n. Function f is said to be 'the big O of a function g ' (notation $f(n) = \mathcal{O}(g(n))$), if there is an N and a constant C such that for all $n \geq N$: $f(n) \leq Cg(n)$, see Wilf [127].

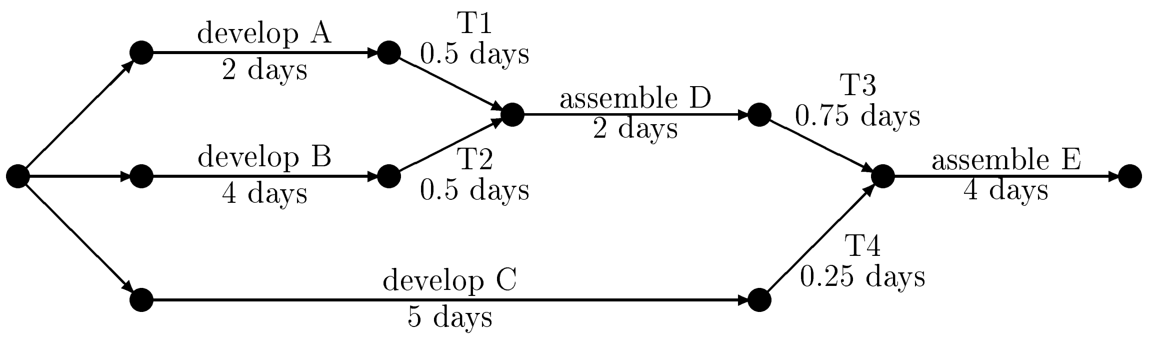


Figure 3.2: An activity network

the Dijkstra algorithm to calculate the shortest path between two nodes in a graph (see [37]). It turns out that the algorithmic efficiency of our method is sufficient for the applications we have in mind.

The most serious drawback of the ATCFN method is that it only produces statements about the arrival time of the first token in a place. In general, we are interested how the system performs under a specific workload and therefore equally interested in the subsequent tokens. We usually also want to verify dynamic properties such as liveness and boundedness. This is the reason we have developed a more general analysis method, which is described in section 3.3.

3.2.1 Application to project engineering

Although the ATCFN method has a number of serious drawbacks, it can be used successfully in the discipline called *project engineering* (Whitehouse [125], Lock [77]). Project engineering, also known as project planning, is concerned with the problem of developing and supervising project plans. We start with a short introduction to the main techniques used in this discipline, followed by an example showing the application of interval timed coloured Petri nets to project planning.

Network planning is an established technique for project planning. In general, it used when a project becomes too complex to plan it just by intuition. There are three basic network types which are used for project planning: *activity networks*, *event networks* and *precedence networks* (see Price [105]).
In an activity network, *activities* (or tasks) are represented by arcs each beginning and ending in an identifiable node of the network. These nodes are called *events* and are represented by circles or vertexes (do not confuse these events with events in an ITCPN). Events are instantaneous and activities are time consuming (i.e. they have a time duration). Figure 3.2 shows an activity network. The nodes representing an event have an *AND/AND* logic, i.e. an event is realized when all input activities have terminated, at this time each of its output activities can start.
For an event network, the interpretation differs from an activity network. Arcs

represent events, circles represent *milestones*. Now time is associated with events. Since the semantics of activity networks and event networks are nearly identical (except for the terminology), we will concentrate on activity networks.

In a precedence network an activity is represented by a node and arrows are used to define the relations between activities. There are four types of relations, i.e. finish-to-start, start-to-finish, finish-to-finish and start-to-start (see Lock [77]). A start-to-finish relationship between two activities A and B means that B cannot finish until a given time after the start of the preceding activity A. Note that it is possible to transform a precedence network into an equivalent activity network.

Two widespread network planning systems are the *CPM* (*Critical Path Method*) system and the *PERT* (*Program Evaluation and Review Technique*). They are both based on activity networks. In a PERT-network, the time duration of an activity is specified by an optimistic estimate, a pessimistic estimate and a most likely estimate.

An event is called a *start event* if there is no input arc. Events without output arcs are called *end events*. In general, a planning network is acyclic and it is defined in such a way that it has one start event and one end event.

The *critical path* in a planning network is the longest path from the start event to the end event. The project duration is given by the length of this critical path. The critical path can be calculated using a *forward calculation* (an activity starts at the time where all previous activities have finished) or *backward calculation* (an activity ends at the time where one of next activities has to start). A forward calculation produces the *earliest event time* of all events, a backward calculation produces the *latest event time* of all events. The critical path of the example shown in figure 3.2 includes the activities *develop B*, *T2*, *assemble D*, *T3* and *assemble E*. The length of the critical path is 11.25 days. Note that for each event on the critical path, the earliest event time equals the latest event time.

The term *float time* (or slack time) is used to describe the amount of *extra* time available for the completion of an activity. There are various kinds of float time, e.g. *total float*, *free float*, *independent float*. These float times are calculated using a forward and backward calculation. For more information on network planning, see [125], [105], [77] and [98].

Interval timed coloured Petri nets are a generalization of the classical activity networks in the sense that they allow for the definition of optimistic and pessimistic estimates of the time durations and in the sense that there are *AND/AND* nodes (transitions) and *OR/OR* nodes (places). In [98], Pagnoni discusses the application of (untimed) Petri nets to project planning.

It is easy to specify an activity network in terms of an ITCPN. An event in a planning network corresponds to a transition in an ITCPN, an activity corresponds to a place. In other words, replace the nodes in the plan by transition bars and the arcs by places connecting two transitions, i.e. precedence relations are represented

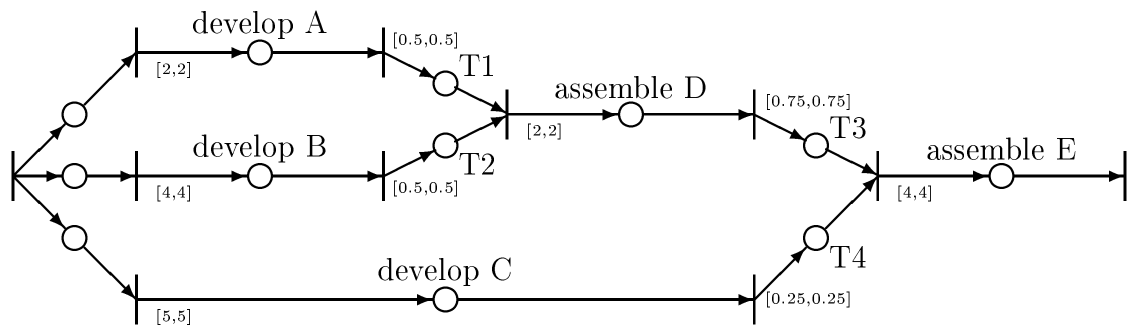


Figure 3.3: An ITCPN representing an activity network

by input and output places. Figure 3.3 shows the ITCPN net corresponding to the the activity net shown in figure 3.2.
An ITCPN constructed this way contains no circuits and has one transition without input places (start event) and one transition without any output places (end event). The transition without the input places fires once (at time 0), this can be modelled by an input place with initially one token with timestamp 0. Note that the interval timed coloured Petri nets constructed like this are acyclic marked graphs (see section 2.5). These nets satisfy the conditions of theorem 3. A forward calculation can be done by applying method ATCFN, the results are upper and lower bounds for the earliest event time. By redirecting of all the arcs in the ITCPN (associate the delay of an activity with the corresponding output arc of the transition which represents the succeeding event), such a calculation produces upper and lower bounds for the latest event time. Therefore, it is possible to calculate various kinds of float times.

Since we use interval timing rather than deterministic delays, we are able to model activities of which the durations are not precisely known. PERT also allows for the modelling of uncertainty. PERT needs three estimates to describe the duration of each activity: an optimistic time, the most likely time and the pessimistic time ([77], [125]). PERT uses these three estimates to specify a beta distribution. Based on this distribution, PERT calculates the average and variance of the duration of the corresponding activity. This information is used to calculate things like the expected (or variance of the) length of the critical path. Note that these results differ from the results produced by the ATCFN method which calculates the upper and lower bound of the length of the critical path.

The traditional network planning techniques, like PERT, do not allow for the representation of 'alternatives', 'choices' and 'cycles'. With an 'alternative' we mean: an event is realized if one (or several) of its input activities terminate. A 'choice' situation is such that if an event is realized, only one of the output activities will start. A 'cycle' is necessary to specify the repeated execution of a set of activities, this way it is possible to represent repetitive schedules (e.g. iterative processes).
Interval timed coloured Petri nets allow for the representation of these aspects. Nets

containing ‘alternatives’, i.e. places with multiple input arcs, can be analysed with the ATCFN method. If a net contains ‘choices’ (conflicts) or ‘cycles’ (circuits), then we have to use one of the analysis methods presented in the remainder of this chapter.

3.3 Method MTSRT

Although the ATCFN method can be used to analyse nets originating from specific application domains (e.g. project planning), it does not meet the requirements set by the systems we want to analyse.
The systems we are interested in often have a behaviour characterized by the words ‘choice’ and ‘repetition’. Consider for example the ITCPN shown in figure 3.4. This ITCPN models two parallel machines A and B, both capable of doing some operation X. Jobs, requiring an operation X, enter the system via place p_1 and leave the system via place p_2 the moment their operation has been completed. Note that the machines share an input buffer (p_1), i.e. a job visits the first available machine. As long as there are jobs waiting in the input buffer, both machines are active.
Place p_1 is called a *conflict place*, because this place has two output arcs. If both machines are free, the next job to be processed selects one of the machines in a non-deterministic manner, i.e. some non-deterministic ‘choice’ has to be made.
The system is also a ‘repetitive’ system, because the machines have to process a number of jobs (e.g. 50 jobs). Therefore, we are interested in the completion time of the n^{th} job, i.e. $\mathcal{EAT}_n(s, p_2)$ and $\mathcal{LAT}_n(s, p_2)$. We are also interested in performance measures like the occupation rate of a machine, i.e. $\mathcal{LOR}$ and $\mathcal{HOR}$. Since the ATCFN method is not suitable for the analysis of these systems, we have developed more powerful methods like the *Modified Transition System Reduction Technique* (MTSRT).

The MTSRT technique is related to the reachability analysis method for usual Petri nets (e.g. Peterson [100]) and is presented in the following.
The transition system $\langle S, R \rangle$ describing the semantics of an ITCPN (see section 2.4.1) defines a so-called *reachability tree*. The root of this tree is the initial state s_1. This root is connected to a number of states $s_{11}, s_{12}, s_{13}, ..$ reachable from s_1 by the firing of some transition, i.e. $\{s_{11}, s_{12}, s_{13}, ..\} = R(s_1)$. These states are called the ‘successors’ (or children) of the root. Every state s_{1i} in $R(s_1)$ is connected to the root and the states reachable from s_{1i}, i.e. its successors $R(s_{1i})$. Repeating this process produces the graphical representation of the reachability tree, see figure 3.5. Such a reachability tree contains all relevant information about the dynamic behaviour of the system. If we are able to generate this tree, we can answer ‘any’ kind of question about the behaviour of the system, for example the performance measures defined in chapter 2.

Several authors present analysis methods based on the generation of (a part) of the reachability tree. In [133], Zuberek proposes such an analysis method, this method

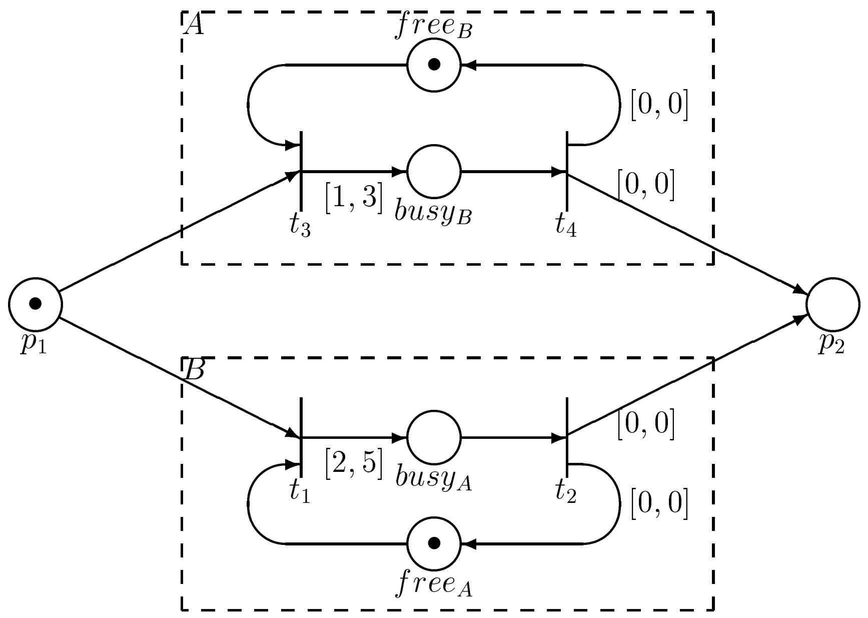


Figure 3.4: Two parallel machines

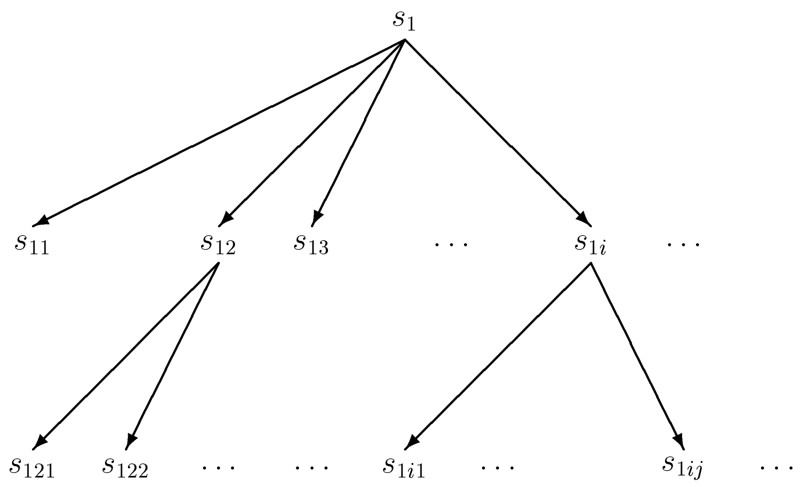


Figure 3.5: A reachability tree

is based on a model with time in transitions and a deterministic firing duration.
In [17] and [16], Berthomieu et al. propose a method to analyse Merlin's timed Petri nets. Although this method uses quite different mathematical techniques, there are some similarities with our MTSRT method. Therefore, they will be compared later.

Other authors have presented analysis techniques for the efficient calculation of a reachability tree of an *untimed coloured* Petri net (e.g. [120], [71], [67], [30]). These techniques are only appropriate if the number of reachable states is finite or if the

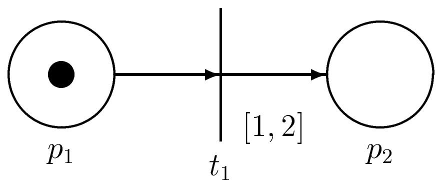


Figure 3.6: An ITCPN

set of reachable states has a special structure.

In general the number of reachable states of an ITCPN (given an initial state) is infinite. This is mainly caused by the fact that we use interval timing. Consider an enabled transition. In general, there is an infinite number of allowed firing delays, all resulting in a different state. Look, for example, at the ITCPN shown in figure 3.6 and suppose that the initial state is such that there is one token in p_1 with timestamp 0. If $TS = \mathbb{R}^+ \cup \{0\}$, then the number of successors of this state is infinite, because all states with one token in p_2 having a timestamp $x \in [1, 2]$ are reachable. It may seem unreasonable that this simple example corresponds to a reachability tree with an infinite number of states. This is the reason we developed the Modified Transition System Reduction Technique. This technique generates the reachability tree and uses, for computational reasons only, an alternative transition system, called the *modified transition system* $\langle \overline{S}, \overline{R} \rangle$. In a sense, this modified transition system gives alternative semantics. The main difference between this transition system and the original one is the fact that we attach a time *interval* to every token instead of a timestamp, i.e. $\overline{S} = Id \not\rightarrow (P \times INT)$.

We will show that, using these semantics, it is possible to calculate the set of reachable states (or at least a relevant subset). The MTSRT method uses the modified transition system to generate (a part of) the reachability tree. Since, the reachability tree of the modified transition system is much smaller and more coarsely grained than the original one, we call it the *reduced reachability tree*. Every state in the reduced reachability tree corresponds to a (infinite) number of states in the reachability tree of the original model. One may think of these states as *equivalence* or *state classes*. One state class $\overline{s} \in \overline{S}$ corresponds to the set of all states being a specialization of $\overline{s}$, i.e. $\{s \in S \mid s \lhd \overline{s}\}$. Informally speaking, state classes are defined as the union of 'similar' states having the same token distribution (marking) but different timestamps (within certain bounds).

Note that it is not our objective to define new semantics, the semantics given in section 2.4.1 specify the meaning of an ITCPN correctly. We use the modified transition system only for reasons of efficiency. However, calculating the reduced reachability tree only makes sense if the reduced reachability tree can be used to

deduce properties of the reachability tree which gives the semantics of the ITCPN. The modified transition system described in the next section has been developed with this objective in mind.
In section 3.3.2, we will show how these two transition systems relate to each other. We will see that the process described by the modified transition system differs from the process described by the original transition system. Nevertheless, we will see that we can use the modified transition system to answer questions about the original transition system and, therefore, about the behaviour of the ITCPN.

3.3.1 The modified transition system

The modified transition system $\langle \overline{S}, \overline{R} \rangle$ is similar to the transition system describing the semantics of an ITCPN. The main difference is the fact that the modified transition system associates a *time interval* with each token rather than a *timestamp*. As a consequence the *state space* is defined as follows:

$$\overline{S} \;=\; Id \not\to (CT \times INT) \tag{3.1}$$

If $s \in \overline{S}$, then $dom(s)$ is the set of token labels corresponding to the tokens in the net. If $i \in dom(s)$, then $s(i)$ is a triplet representing the *position*, *value* and *time interval* of the corresponding token. The time interval of a token represents the upper and lower bound for the time it becomes available.

We want to use this state space for reasons of computational efficiency. On the other hand, we are interested in a transition system which resembles the original transition system given in section 2.4.1 as much as possible, because we want to use the modified transition system to analyse the behaviour of the ITCPN (which is described by the original transition system). Therefore, we define the transition relation $\overline{R}$ as follows.

For convenience we define a number of functions to refer to a specific aspect of a token.

Definition 25
For $q \in CT \times INT$ we define:

$$\begin{aligned}
place(q) &= \pi_1(\pi_1(q)) \\
value(q) &= \pi_2(\pi_1(q)) \\
time(q) &= \pi_2(q) \\
time^{min}(q) &= \pi_1(\pi_2(q)) \\
time^{max}(q) &= \pi_2(\pi_2(q))
\end{aligned}$$

If $s \in \overline{S}$ and $i \in dom(s)$ is the label of a token in this state, then the arrival time (the time the token becomes available) is between $time^{min}(s(i))$ and $time^{max}(s(i))$.

We define $\overline{E}$ to be the *event set* of the modified transition system:

$$\overline{E} \;=\; T \times \overline{S} \times \overline{S} \tag{3.2}$$

An event changes a state into a new state, described by the transition relation. An event $e \in \overline{E}$ is a triplet indicating the transition that fires $\pi_1(e)$, the tokens which are consumed $\pi_2(e)$ and the tokens which are produced $\pi_3(e)$.

$\overline{AE}(s) \subseteq \overline{E}$ is the set of *allowed events* in state $s \in \overline{S}$. An allowed event $e \in \overline{AE}(s)$ satisfies five conditions, which are similar to the conditions given on page 39. In the original transition system, tokens are selected in order of their timestamps. The modified transition systems associates a time interval with each token (instead of a timestamp). Therefore, we define the relation $\leq_i$ to compare intervals, i.e. to select tokens in order of their timestamps.

Definition 26 ($\leq_i$)
If $v, w \in INT$, then: $v \leq_i w \;\equiv\; (\pi_1(v) \leq \pi_1(w)) \;\wedge\; (\pi_2(v) \leq \pi_2(w))$

Note that $\leq_i$ defines a partial ordering, because $\leq_i$ is reflexive, antisymmetric and transitive. Sometimes we use the notation $v <_i w$ to denote that $v \leq_i w$ and $v \neq w$. We use figure 3.7 to illustrate this relation, an interval is represented by a line segment, the lower bound of the interval is represented by a left bracket ([), the upper bound of the interval is represented by a right bracket (]). In figure 3.7(a) and (b) we see situations where $v \leq_i w$ and $w \leq_i v$ respectively. The intervals of figure 3.7(c) and (d) are incomparable, i.e. $\neg(v \leq_i w)$ and $\neg(w \leq_i v)$.
Suppose a place p contains two tokens with time intervals as in figure 3.7(c), then it is impossible to decide which token is consumed first, because w contains timestamps smaller than any timestamp in v and timestamps larger than any timestamp in v. If a transition t having this place as its input place ($I_t(p) = 1$) is enabled, then there are at least two allowed events, one consuming the token with time interval v and one consuming the token with time interval w. On the other hand, if place p contains two tokens having the same value, one with time interval v and the other with time interval w such that $v <_i w$, then if suffices to consider the event consuming the token with timestamp v (see lemma 27 in the appendix of this chapter). In other words: tokens having the same value are consumed in non-descending order rather than ascending order. We do this, because $\leq_i$ is not a total ordering (see appendix).

To discard the timestamps of the tokens in a state, we define the function $\overline{untime} \in \overline{S} \rightarrow (Id \not\rightarrow CT)$. If $s \in \overline{S}$, then:

$$\overline{untime}(s) \;=\; \lambda_{i \in dom(s)}\, \langle place(s(i)), value(s(i)) \rangle \tag{3.3}$$

Now we can formalize $\overline{AE}(s)$, the set of allowed events in state $s \in S$. An allowed

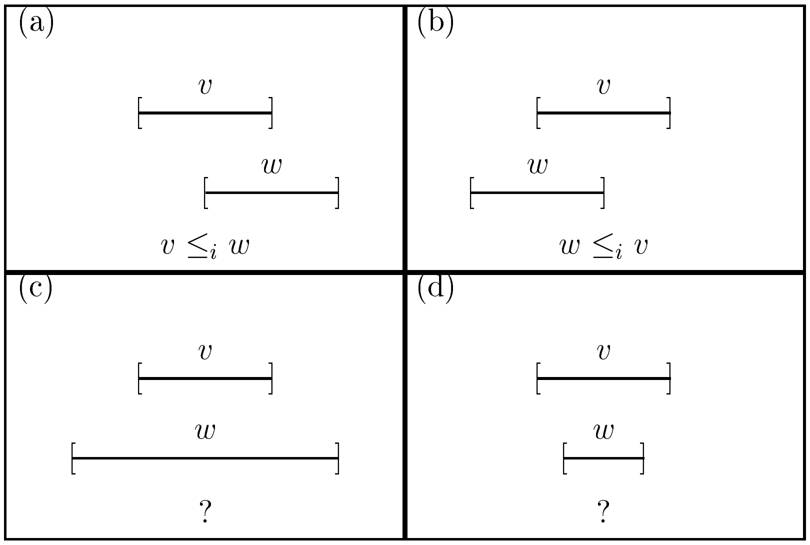


Figure 3.7: Comparing two intervals v and w

event $e \in \overline{AE}(s)$ satisfies 5 conditions. The first condition is about the requirement that consumed tokens have to exist. The transition that fires consumes the correct number of tokens from the input places (condition (b)). Tokens in the same place having the same value are consumed in non-descending order (condition (c)). Produced tokens bear a unique label, condition (d) checks whether the label of a produced token does not exist already. The delay interval of a produced token is as specified by function F (condition (e)).

$$\begin{aligned}
\overline{AE}(s) \quad = \quad & \{\langle t, q_{in}, q_{out}\rangle \in \overline{E} \mid q_{in} \subseteq s \land && (3.4a)\\
& I_t = \lambda_{p \in P}\ \#\{i \in dom(q_{in}) \mid place(s(i)) = p\} \land && (3.4b)\\
& \forall_{i \in dom(q_{in})} \forall_{j \in dom(s) \setminus dom(q_{in})}\ (\ place(s(i)) = place(s(j)) \land && \\
& \quad value(s(i)) = value(s(j))\ ) \Rightarrow \neg(time(s(j)) <_i time(s(i))) \land && (3.4c)\\
& dom(q_{out}) \cap dom(s) = \emptyset \land && (3.4d)\\
& \mathcal{SB}(q_{out}) = F_t(\mathcal{SB}(\overline{untime}(q_{in})))\ \} && (3.4e)
\end{aligned}$$

Each of the requirements (3.4a), (3.4b), .. (3.4e) corresponds to one of the conditions mentioned before. The delay intervals of the produced tokens are given by the expression $F_t(\mathcal{SB}(\overline{untime}(q_{in})))$. Because the domain of F_t is a subset of $\mathbb{B}(CT)$, we have to use the function $\overline{untime}$ to omit the timestamps of the consumed tokens. The function $\mathcal{SB}$ is needed, because the function F is defined in terms of bags and the transition system uses partial functions to denote bags.

The point of time a token becomes available is specified by an interval, therefore it is impossible to specify *the* event time of an event. However, it is possible to give an upper and lower bound for the *event time* of an event $e \in \overline{E}$:

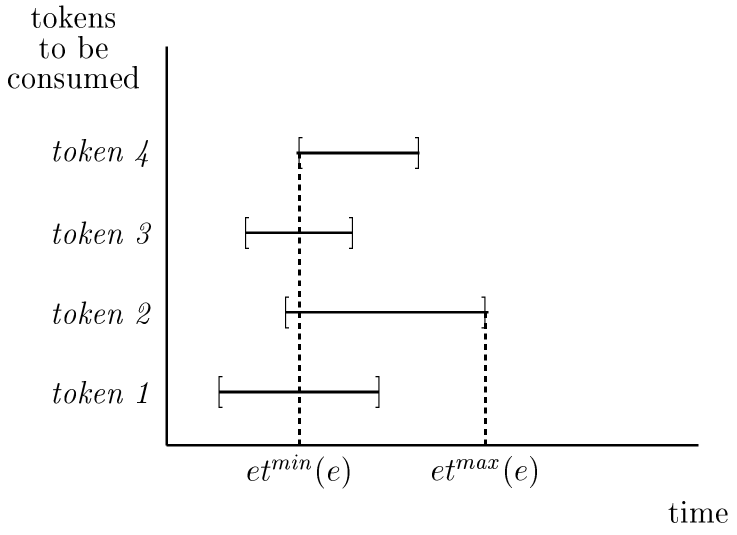


Figure 3.8: $et^{min}(e)$ and $et^{max}(e)$

$$et^{min}(e) = \max_{i \in dom(\pi_2(e))} time^{min}(\pi_2(e)(i)) \tag{3.5}$$
$$et^{max}(e) = \max_{i \in dom(\pi_2(e))} time^{max}(\pi_2(e)(i)) \tag{3.6}$$

This is illustrated in figure 3.8 where the time intervals of the tokens to be consumed are represented by horizontal line segments. The event time of an event e in isolation is between $et^{min}(e)$ and $et^{max}(e)$.

The *transition time* is the event time of the first event to occur, i.e. the minimum of the event times of the allowed events. Since the event time of an event in the modified transition system is characterized by an interval, the transition time of a state $s \in \overline{S}$ is also characterized by an interval:

$$tt^{min}(s) = \min_{e \in AE(s)} et^{min}(e) \tag{3.7}$$
$$tt^{max}(s) = \min_{e \in AE(s)} et^{max}(e) \tag{3.8}$$

This means that the first event in state s will occur between $tt^{min}(s)$ and $tt^{max}(s)$, this is illustrated in figure 3.9. An allowed event $e \in \overline{AE}(s)$ may occur, if and only if, $et^{min}(e) \leq tt^{max}(s)$. If it occurs, then it occurs at a time between $et^{min}(e)$ and $tt^{max}(s)$.

For an allowed event $e \in \overline{AE}(s)$, the time intervals in $\pi_3(e)$ correspond to the *firing delays* of the produced tokens. Therefore, we have to rescale the (relative) intervals

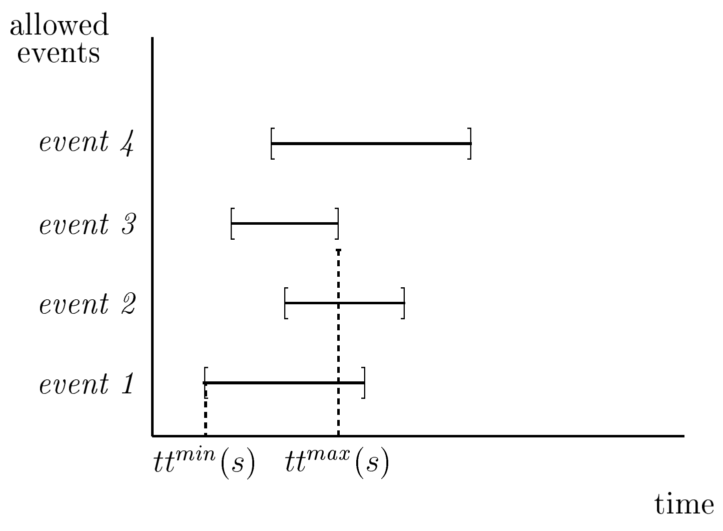


Figure 3.9: $tt^{min}(s)$ and $tt^{max}(s)$

of these produced tokens. For this purpose we define the function $\overline{scale}$. If $q \in \overline{S}$ and $x, y \in TS$, then:

$$\begin{aligned}\overline{scale}(q,x,y) \;=\; \lambda_{i \in dom(q)} \;& \langle\langle place(q(i)), value(q(i))\rangle,\\ & \langle time^{min}(q(i)) + x, time^{max}(q(i)) + y\rangle\rangle \end{aligned} \tag{3.9}$$

This function is used to add $et^{min}(e)$ to the lower bound of each delay interval and to add $tt^{max}(s)$ to the upper bound of each delay interval.

Finally, the transition relation of the modified transition system is defined by:

$$s_1 \overline{R} s_2 \;\equiv\; \exists_{\substack{e \in \overline{AE}(s_1)\\ et^{min}(e) \leq tt^{max}(s_1)}} \; s_2 = (s_1 \setminus \pi_2(e)) \;\cup\; \overline{scale}(\pi_3(e), et^{min}(e), tt^{max}(s_1)) \tag{3.10}$$

for $s_1, s_2 \in \overline{S}$.

The complete transition system is summarized below.

The modified transition system

An ITCPN $= (P, V, T, I, O, F, TS)$ defines a *modified transition system* $\langle \overline{S}, \overline{R} \rangle$, with a state space $\overline{S}$ and a transition relation $\overline{R}$:

- $\overline{S} = Id \not\rightarrow (CT \times INT)$, the state space
- $\overline{E} = T \times \overline{S} \times \overline{S}$, event set
- $\overline{untime}(s) = \lambda_{i \in dom(s)} \; \langle place(s(i)), value(s(i))\rangle$, deletes the time intervals in state $s \in \overline{S}$

- $\overline{AE}(s) =$

$$\begin{aligned} \{ \ & \langle t, q_{in}, q_{out} \rangle \in \overline{E} \mid q_{in} \subseteq s \ \wedge \\ & I_t = \lambda_{p \in P} \ \#\{ i \in dom(q_{in}) \mid place(s(i)) = p \} \ \wedge \\ & \forall_{i \in dom(q_{in})} \forall_{j \in dom(s) \setminus dom(q_{in})} \ (\ place(s(i)) = place(s(j)) \ \wedge \\ & \qquad value(s(i)) = value(s(j)) \ ) \ \Rightarrow \ \neg(time(s(j)) <_i time(s(i)) \ \wedge \\ & dom(q_{out}) \cap dom(s) = \emptyset \ \wedge \\ & \mathcal{SB}(q_{out}) = F_t(\mathcal{SB}(\overline{untime}(q_{in}))) \ \} \ , \end{aligned}$$

  the set of allowed events in state $s \in \overline{S}$

- $et^{min}(e) = \max_{i \in dom(\pi_2(e))} time^{min}(\pi_2(e)(i))$, lower bound event time of $e \in \overline{E}$
- $et^{max}(e) = \max_{i \in dom(\pi_2(e))} time^{max}(\pi_2(e)(i))$, upper bound event time of $e \in \overline{E}$
- $tt^{min}(s) = \min_{e \in \overline{AE}(s)} et^{min}(e)$, lower bound transition time in $s \in \overline{S}$
- $tt^{max}(s) = \min_{e \in \overline{AE}(s)} et^{max}(e)$, upper bound transition time in $s \in \overline{S}$
- $\overline{scale}(q, x, y) = \lambda_{i \in dom(q)} \langle \pi_1(q(i)), \langle time^{min}(q(i)) + x, time^{max}(q(i)) + y \rangle \rangle$, scales timestamps, $q \in \overline{S}$ and $x, y \in TS$
- Finally, the transition relation $\overline{R}$ is defined as follows. If $s_1, s_2 \in \overline{S}$, then:

$$s_1 \overline{R} s_2 \equiv \exists_{\substack{e \in \overline{AE}(s_1) \\ et^{min}(e) \leq tt^{max}(s_1)}} \quad s_2 = (s_1 \setminus \pi_2(e)) \ \cup \ \overline{scale}(\pi_3(e), et^{min}(e), tt^{max}(s_1))$$

Note the resemblance with the original transition system described in section 2.4.1. Comparing the two transition systems shows that all differences stem from the fact that the modified transition system associates a time interval (instead of a timestamp) with each token. As a result of these intervals, the event time of an event and the transition time of a state are both characterized by an upper and lower bound, etc.

To give an impression of the modified transition system, consider the net shown in figure 3.10. Initially, there is one token in place $p1$ with an interval of $[0, 3]$, there is one token in $p2$ with an interval of $[2, 5]$ and there is one token in $p3$ with an interval of $[4, 6]$. Note that this state in the modified transition system (i.e. a state class) corresponds to an infinite number of states in the original model, for instance the state with a token in $p1$ with timestamp 2.4 and a token in $p2$ with timestamp π and a token in $p3$ with timestamp $31/6$.
There are two allowed events, event e_1 is the firing of $t1$ while consuming the tokens in $p1$ and $p2$, event e_2 is the firing of $t2$ while consuming the tokens in $p2$ and $p3$. The event time of e_1 is between 2 ($et^{min}(e_1)$) and 5 ($et^{max}(e_1)$), the event time of e_2 is between 4 ($et^{min}(e_2)$) and 6 ($et^{max}(e_2)$). All events having a lower bound for the event time (et^{min}) smaller than or equal to the upper bound of the transition time

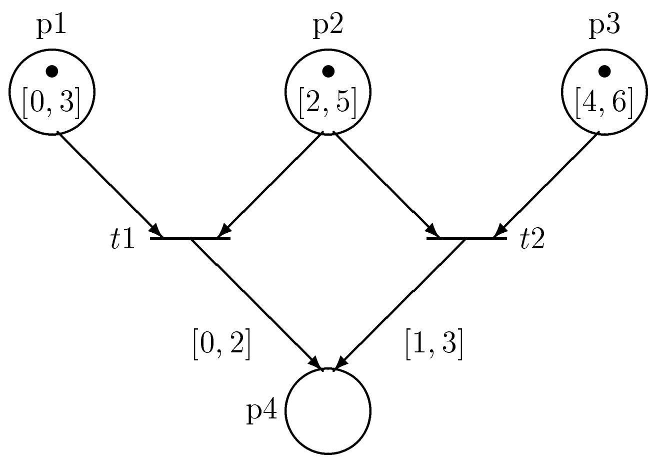


Figure 3.10: An example used to illustrate the modified transition system

(tt^{max}) can happen. If e_1 occurs, it will be between 2 ($et^{min}(e_1)$) and 5 ($tt^{max}(s)$). If e_2 occurs, it will be between 4 ($et^{min}(e_2)$) and 5 ($tt^{max}(s)$). In both cases a token is produced for place p4. There are two possible terminal states: one with a token in $p3$ and $p4$ and one with a token in $p1$ and $p4$. In the first case the time interval of the token in $p4$ is $[2, 7]$, because the delay interval of a token produced by $t1$ is $[0, 2]$. In the second case the time interval of the token in $p4$ is $[5, 8]$. Using intervals rather than timestamps prevented us from having to consider all possible delays in the intervals $[0, 2]$ and $[1, 3]$, i.e. it suffices to consider upper and lower bounds. Nevertheless, we will see that the process described by the modified transition system differs from the process described by the original transition system.

In the remainder of this chapter, we assume that $\langle S, R\rangle$ is the transition system describing the semantics of an ITCPN (P, V, T, I, O, F, TS) and $\langle \overline{S}, \overline{R}\rangle$ is the corresponding modified transition system. Symbols superscripted by a horizontal line are associated with the modified transition system, this to avoid confusion.
For example, if $A \subseteq \overline{S}$, then $\overline{R}(A)$ is the set of all states reachable by firing one transition in a state in A. $\overline{RS}(A) = \cup_{n\in\mathbb{N}} \overline{R}^n(A)$ is the set of all states reachable by firing an arbitrary number of transitions (when starting in a state in A). $\overline{S}^T = \{s \in \overline{S} \mid \overline{R}(s) = \emptyset\}$, the set of *terminal states.*
The *process* which corresponds to the modified transition system and a set of initial states $A \subseteq \overline{S}$, is described by the set of all possible *paths.* Recall, a path is a sequence of states such that any successive pair belongs to the transition relation of the modified transition system. A path starts in an initial state and either it is infinite or it ends in a terminal state (see definition 5).
The other properties and performance measures defined in chapter 2 are also defined for the modified transition system in a straightforward manner. To distinguish these performance measures from the original ones, we also superscript them by a horizontal line.

Most of the theorems of chapter 2, based on the original transition system, are also valid for the modified transition system. Consider for example the theorem about the ‘monotonicity of time’ (theorem 1), i.e. the property that time can only move forward. The following theorem shows that the upper and lower bounds of the transition times in the modified transition system are also ‘non-decreasing’.

Theorem 5
Let $\langle \overline{S}, \overline{R} \rangle$ be the modified transition system of an arbitrary ITCPN. For any state $s \in \overline{S}$, any path $\sigma \in \overline{\Pi}(s)$ and any $i, j \in dom(\sigma)$ such that $i \leq j$, we have: $tt^{min}(\sigma_i) \leq tt^{min}(\sigma_j)$ and $tt^{max}(\sigma_i) \leq tt^{max}(\sigma_j)$.

Proof.
First, we prove that for all $s_1, s_2 \in \overline{S}$ with $s_1 \overline{R} s_2$: $tt^{min}(s_1) \leq tt^{min}(s_2)$ and $tt^{max}(s_1) \leq tt^{max}(s_2)$.
Because $s_2 \in \overline{R}(s_1)$, there exists an event $e \in \overline{AE}(s_1)$ such that $et^{min}(e) \leq tt^{max}(s_1)$ and $s_2 = (s_1 \setminus \pi_2(e)) \cup \overline{scale}(\pi_3(e), et^{min}(e), tt^{max}(s_1))$. The definition of $\overline{scale}$ tells us that the lower bound of the produced token is at least $et^{min}(e)$ and the upper bound is at least $tt^{max}(s_1)$. Hence, for all new events $h \in \overline{AE}(s_2) \setminus \overline{AE}(s_1)$ we find that $et^{min}(h) \geq et^{min}(e) \geq tt^{min}(s_1)$ and $et^{max}(h) \geq tt^{max}(s_1)$. All events that where already enabled also have a lower bound event time of at least $tt^{min}(s_1)$ and an upper bound event time of at least $tt^{max}(s_1)$. By the definition of tt^{min} and tt^{max} we conclude that $tt^{min}(s_1) \leq tt^{min}(s_2)$ and $tt^{max}(s_1) \leq tt^{max}(s_2)$.
Note that $\sigma_i \overline{R}^{j-i} \sigma_j$. Using induction in $n \in \mathbb{N}$ it is easy to prove that $\sigma_i \overline{R}^{n} \sigma_j$ implies that $tt^{min}(\sigma_i) \leq tt^{min}(\sigma_j)$ and $tt^{max}(\sigma_i) \leq tt^{max}(\sigma_j)$.
$\Box$

3.3.2 Using the modified transition system

We have developed the modified transition system for computational reasons. However, calculating the reduced reachability tree only makes sense if the reduced reachability tree can be used to deduce properties of the reachability tree which gives the semantics of the ITCPN. Therefore, we investigate the relation between the two transition systems. Examples indicate that such a relation exists. Since the original transition system describes the semantics of an ITCPN, it is necessary to establish a formal relation between the two transition systems. Without this formal relationship we are unable to answer questions about the ITCPN using the modified transition system.
It is easy to see that the two transition systems are not equivalent. Moreover, there is no sensible morphism between these two transition systems. We will use a small example to show this. Consider a net composed of one place p and without transitions, $V_p = \{\text{‘}\texttt{signal}\text{’}\}$ and $TS = \mathbb{R}^+ \cup \{0\}$. The corresponding original and modified transition system are given by $\langle S, R \rangle$ and $\langle \overline{S}, \overline{R} \rangle$ respectively. If $x \in TS$ and $\langle y, z \rangle \in INT$, then $s = \{\langle 1, \langle\langle p, \text{‘}\texttt{signal}\text{’}\rangle, x\rangle\rangle\} \in S$ and $\overline{s} = \{\langle 1, \langle\langle p, \text{‘}\texttt{signal}\text{’}\rangle, \langle y, z\rangle\rangle\rangle\} \in \overline{S}$. In this case our intuition says that s and $\overline{s}$ are ‘related’ if and only if $y \leq x \leq z$. There is no morphism capable of expressing this relation, because s corresponds to a lot of states in $\overline{S}$ and $\overline{s}$ corresponds to a lot of states in S. This is a direct result of the fact that we use interval timing.
However, it is possible that there exists a useful similarity relationship. The small example shows that it is sensible to use the specialization concept defined in section 2.4.1 to relate the states of the two transition systems. Recall that for $s \in S$ and

$\overline{s} \in \overline{S}$, s is a specialization of $\overline{s}$ (notation: $s \lhd \overline{s}$), if and only if, there exists a bijective function $f \in dom(s) \rightarrow dom(\overline{s})$ such that every token with label $i \in dom(s)$ corresponds to a token with label $f(i) \in dom(\overline{s})$ that is in the same place, has the same value and has an interval containing the timestamp of i. Based on this concept, we define two similarity relations. See section 2.3 for a formal definition of similarity.

Definition 27 (Soundness)
For an ITCPN (P, V, T, I, O, F, TS), the combination of the corresponding original transition system $X = \langle S, R \rangle$ and modified transition system $Y = \langle \overline{S}, \overline{R} \rangle$ is called *sound*, if and only if, Y is similar to X with respect to the specialization relation $\{\langle s, \overline{s} \rangle \in S \times \overline{S} \mid s \lhd \overline{s}\}$.

Definition 28 (Completeness)
For an ITCPN (P, V, T, I, O, F, TS), the combination of the corresponding original transition system $X = \langle S, R \rangle$ and modified transition system $Y = \langle \overline{S}, \overline{R} \rangle$ is called *complete*, if and only, if X is similar to Y with respect to the relation $\{\langle \overline{s}, s \rangle \in \overline{S} \times S \mid s \lhd \overline{s}\}$.

Informally speaking, soundness means that states reachable in the original model are also reachable in the modified transition system based on state classes. Completeness means that all transitions possible in $\langle \overline{S}, \overline{R} \rangle$ are also possible in $\langle S, R \rangle$.
If both similarity relations hold, we speak about bisimilarity with respect to specialization. Since bisimilarity w.r.t. the specialization relation is a rather strong property, this property *would* have been very useful.
Unfortunately, completeness does not always hold, this is caused by the fact that *dependencies* between tokens are not taken into account. Consider for example the net shown in figure 3.11. Suppose there is one token in $p1$ with a time interval $[0, 1]$ and the other places are empty. In this case t fires between time 0 ($et^{min}(e)$) and time 1 ($tt^{max}(s)$). The next state in the modified transition system will be the state with one token in $p2$ (with interval $[1, 3]$) and one token in $p3$ (with interval $[3, 5]$). This suggests that it is possible to have a token in $p2$ with timestamp 1 and a token in $p3$ with timestamp 5. However, this is not possible (in the original transition system), because these timestamps are related (i.e. they where produced at the same time).

Fortunately, for any ITCPN the soundness property holds:

Theorem 6 (Soundness)
For any ITCPN (P, V, T, I, O, F, TS), we have that the combination of the corresponding original transition system $X = \langle S, R \rangle$ and modified transition system $Y = \langle \overline{S}, \overline{R} \rangle$, is sound.

Proof.
Let $s_1 \in S$ and $\overline{s}_1 \in \overline{S}$ such that $s_1 \lhd \overline{s}_1$, and $s_2 \in R(s_1)$, see figure 3.12. Now we have to prove that there exists an $\overline{s}_2 \in \overline{R}(\overline{s}_1)$ such that $s_2 \lhd \overline{s}_2$ (see definition 7).

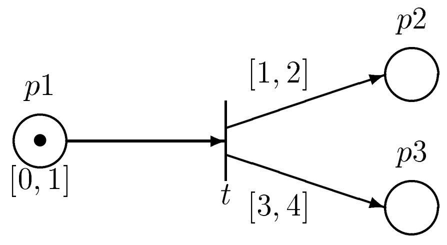


Figure 3.11: Non-completeness caused by dependencies

Since $s_1 \lhd \overline{s}_1$, there exists a specialization function f, i.e. there exists a bijective function $f \in dom(s_1) \rightarrow dom(\overline{s}_1)$ such that every token with label $i \in dom(s_1)$ corresponds to a token with label $f(i) \in dom(\overline{s}_1)$ that is in the same place, has the same value and has an interval containing the timestamp of i.
Because $s_1 R s_2$, there is an event e such that:

(i) $e \in AE(s_1)$

(ii) $et(e) = tt(s_1)$

(iii) $s_2 = (s_1 \setminus \pi_2(e)) \cup scale(\pi_3(e), tt(s_1))$

Define $\overline{e} = \langle \pi_1(e), \overline{s}_1 \restriction f(dom(\pi_2(e))), q \rangle \in \overline{E}$, where $q \in \overline{S}$ such that conditions (3.4d) and (3.4e) on page 84 hold. This is always possible, because condition (3.4e) specifies the labelled bag q precisely (except for the labels) and condition (3.4d) says that the labels have to be 'new'. Note that $\pi_3(e) \lhd q$.
Define $\overline{s}_2 = \overline{s}_1 \setminus \pi_2(\overline{e}) \cup \overline{scale}(\pi_3(\overline{e}), et^{min}(\overline{e}), tt^{max}(\overline{s}_1))$.

Now it suffices to prove that:

(i) $\overline{e} \in \overline{AE}(\overline{s}_1)$

(ii) $et^{min}(\overline{e}) \leq tt^{max}(\overline{s}_1)$

(iii) $s_2 \lhd \overline{s}_2$

(i) Event $\overline{e}$ is an element of $\overline{AE}(\overline{s}_1)$ if it satisfies the five conditions stated in the definition of $\overline{AE}$. All conditions except condition (3.4c) follow directly from the definition of $\overline{e}$ and the fact that $e \in AE(s_1)$. To prove the fact that condition (3.4c) holds, we have to impose additional restrictions on f, however, it is always possible to transform ('massage') f such that (3.4c) holds (see the appendix of this chapter, lemma 27).

(ii) Since $\pi_2(e) \lhd \pi_2(\overline{e})$, we have:

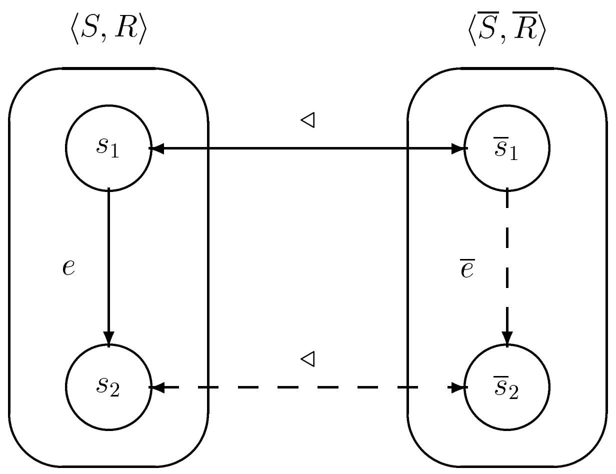


Figure 3.12: The soundness property, i.e. $\langle \overline{S}, \overline{R}\rangle$ is similar to $\langle S, R\rangle$ with respect to the specialization relation $\{\langle s, \overline{s}\rangle \in S \times \overline{S} \mid s \lhd \overline{s}\}$

$et^{min}(\overline{e}) = \max_{i \in dom(\pi_2(\overline{e}))} time^{min}(\pi_2(\overline{e})(i)) \leq \max_{i \in dom(\pi_2(e))} time(\pi_2(e)(i)) = et(e)$
That is, $et^{min}(\overline{e}) \leq et(e)$. It is also easy to verify that: $tt(s_1) \leq tt^{max}(\overline{s}_1)$, because $s_1 \lhd \overline{s}_1$.
Therefore: $et^{min}(\overline{e}) \leq et(e) = tt(s_1) \leq tt^{max}(\overline{s}_1)$.

(iii) From $s_1 \lhd \overline{s}_1$ and the definition of $\pi_2(\overline{e})$ we deduce that: $(s_1 \setminus \pi_2(e)) \lhd (\overline{s}_1 \setminus \pi_2(\overline{e}))$
Since $et^{min}(\overline{e}) \leq tt(s_1) \leq tt^{max}(\overline{s}_1)$ and $\pi_3(e) \lhd \pi_3(\overline{e})$, we have:
$scale(\pi_3(e), tt(s_1)) \ \lhd \ \overline{scale}(\pi_3(\overline{e}), et^{min}(\overline{e}), tt^{max}(\overline{s}_1))$
This implies that $s_2 \lhd \overline{s}_2$.
□

This theorem tells us that if a transition is possible from s_1 to s_2 in the original transition system, there is a corresponding transition in the modified transition system from every state $\overline{s}_1$ that 'covers' s_1.

How are the paths in the modified transition system related to the paths in the original transition system? To investigate this, we also define the specialization concept for paths ($\lhd_\pi$).

Definition 29 (Specialization)
For $\sigma \in \mathbb{N} \not\rightarrow S$ and $\overline{\sigma} \in \mathbb{N} \not\rightarrow \overline{S}$: $\quad \sigma \lhd_\pi \overline{\sigma} \equiv (dom(\sigma) = dom(\overline{\sigma}) \ \wedge \ \forall_{i \in dom(\sigma)} \ \sigma_i \lhd \overline{\sigma}_i)$

Now it is possible to show that soundness also holds for the processes (Π and $\overline{\Pi}$) generated by the two transition systems.

Lemma 14
For all $s_1 \in S$ and $\overline{s}_1 \in \overline{S}$ such that $s_1 \lhd \overline{s}_1$: $\quad \forall_{\sigma \in \Pi(s_1)} \ \exists_{\overline{\sigma} \in \overline{\Pi}(\overline{s}_1)} \ \sigma \lhd_\pi \overline{\sigma}$

Proof.
If σ is an infinite path (i.e. $dom(\sigma) = \mathbb{N}$), then we have to prove that there is a $\overline{\sigma}$ such that $dom(\overline{\sigma}) = \mathbb{N}$ and $\forall_{i \in dom(\sigma)}\ \sigma_i \lhd \overline{\sigma}_i$. Since $s_1 \lhd \overline{s}_1$, we find that $\sigma_0 \lhd \overline{\sigma}_0$. For all $i \geq 0$ take $\overline{\sigma}_{i+1} \in \overline{R}(\overline{\sigma}_i)$ such that $\sigma_{i+1} \lhd \overline{\sigma}_{i+1}$. This is possible, because of the soundness property (theorem 6). If σ is a finite path of length n, then we have to prove that $\overline{\sigma}_{n-1}$ is a terminal state. We know that $R(\sigma_{n-1}) = \emptyset$ and that $\sigma_{n-1} \lhd \overline{\sigma}_{n-1}$. Moreover, $\overline{R}(\overline{\sigma}_{n-1}) = \emptyset$, if and only, if there is no transition enabled, i.e. there is no transition with sufficient tokens on each of its input places. This implies that $\overline{R}(\overline{\sigma}_{n-1}) = \emptyset$, because if $AE(\sigma_{n-1}) = \emptyset$, then $\overline{AE}(\overline{\sigma}_{n-1}) = \emptyset$.
$\Box$

Despite the non-completeness, the soundness property allows us to answer some of the questions stated in section 2.6. We can *prove* that a system has a desired set of properties by proving it for the modified transition system. For example:

Lemma 15
For any $p \in P$, $K \in \mathbb{N}$, $s \in S$ and $\overline{s} \in \overline{S}$ such that $s \lhd \overline{s}$, we have:

$$\forall_{\hat{s} \in \overline{RS}(\overline{s})}\ \#(\hat{s} \upharpoonleft p) \leq K \quad \Rightarrow \quad \forall_{\hat{s} \in RS(s)}\ \#(\hat{s} \upharpoonleft p) \leq K$$

Proof.
If $\hat{s} \in S$, $s' \in \overline{S}$ and $\hat{s} \lhd s'$, then for any $p \in P$: $\#(\hat{s} \upharpoonleft p) = \#(s' \upharpoonleft p)$ (see definition of specialization).
Theorem 6 implies that for any $\hat{s} \in RS(s)$, there exists a $s' \in \overline{RS}(\overline{s})$ such that $\hat{s} \lhd s'$ (we can prove this by induction).
Assume that for all $s' \in \overline{RS}(\overline{s})$: $\#(s' \upharpoonleft p) \leq K$. Now it is easy to see that for any $\hat{s} \in RS(s)$: $\#(\hat{s} \upharpoonleft p) \leq K$, because if there exists a $\hat{s} \in RS(s)$ such that $\#(\hat{s} \upharpoonleft p) > K$, then there also exists a $s' \in \overline{RS}(\overline{s})$ such that $\#(s' \upharpoonleft p) > K$ (i.e. a contradiction).
$\Box$

This lemma states the fact that if the modified transition system indicates that an ITCPN is K-bounded (or safe) for an initial state, then the net is K-bounded (or safe) for that initial state with respect to the original transition system. In other words, we can use the modified transition system to prove boundedness.

We also use the modified transition system to calculate bounds for the arrival times of tokens in a place. Although these bounds are sound (i.e. safe) they do not have to be as tight as possible, because of possible dependencies between tokens (non-completeness). First, we define the earliest and latest arrival time for the modified transition system. To do this we need to define place projection ($\upharpoonleft^{min}$ and $\upharpoonleft^{max}$) for $\overline{S}$.

Definition 30 ($\upharpoonleft^{min}$, $\upharpoonleft^{max}$)
For all $\overline{s} \in \overline{S}$, $p \in P$:
$\overline{s} \upharpoonleft^{min} p = \lambda_{x \in TS}\ \#\{i \in dom(\overline{s}) \mid place(\overline{s}(i)) = p\ \wedge\ time^{min}(\overline{s}(i)) = x\}$
$\overline{s} \upharpoonleft^{max} p = \lambda_{x \in TS}\ \#\{i \in dom(\overline{s}) \mid place(\overline{s}(i)) = p\ \wedge\ time^{max}(\overline{s}(i)) = x\}$

That is, $\overline{s} \upharpoonright^{min} p$ ($\overline{s} \upharpoonright^{max} p$) gives the bag of lower (upper) bounds of the intervals of the tokens in p).

Definition 31 ($\overline{\mathcal{EAT}}_n, \overline{\mathcal{LAT}}_n$)
If $\overline{A} \subseteq \overline{S}$ and $p \in P$, then:

$$\overline{\mathcal{EAT}}_n(A, p) = \min_{\overline{\sigma} \in \overline{\Pi}(\overline{A})} \ \min_{i \in dom(\overline{\sigma})} \ \text{bmin}_n(\overline{\sigma}_i \upharpoonright^{min} p)$$

$$\overline{\mathcal{LAT}}_n(A, p) = \max_{\overline{\sigma} \in \overline{\Pi}(\overline{A})} \ \min_{i \in dom(\overline{\sigma})} \ \text{bmin}_n(\overline{\sigma}_i \upharpoonright^{max} p)$$

The following lemma shows that we can use the modified transition system to deduce bounds for the arrival time of the n^{th} token. In this way we can prove that certain deadlines are met.

Lemma 16
If $A \subseteq S, \overline{A} \subseteq \overline{S}, p \in P$ and $\forall_{s \in A} \ \exists_{\overline{s} \in \overline{A}} \ s \lhd \overline{s}$, then:

- $\overline{\mathcal{EAT}}_n(\overline{A}, p) \leq \mathcal{EAT}_n(A, p)$
- $\overline{\mathcal{LAT}}_n(\overline{A}, p) \geq \mathcal{LAT}_n(A, p)$

Proof.
If $s \lhd \overline{s}$, then $\text{bmin}_n(\overline{s} \upharpoonright^{min} p) \leq \text{bmin}_n(s \upharpoonright p) \leq \text{bmin}_n(\overline{s} \upharpoonright^{max} p)$.
Use these inequalities and lemma 14 to verify the assertion of this lemma.
$\square$

It is also possible to define $\overline{\mathcal{LOR}}$ and $\overline{\mathcal{HOR}}$ in such a way that they have similar properties. In this way the modified transition system can be used to derive 'safe' upper and lower bounds for performance measures like occupation rate and (average) stock levels. Note that if the original transition system and modified transition system would have been bisimilar with respect to the similarity relation (i.e. sound *and* complete), then these bounds would have been as tight as possible.

We have demonstrated that we can use the modified transition system to answer all kinds of questions about the original model. This is only useful if the corresponding reduced reachability tree is finite. In other words, the MTSRT method is unable to answer questions which require the generation of an 'unbounded' reduced reachability tree. However, the following theorem shows that progressive nets can be analysed using the MTSRT method, because the relevant part of the reduced reachability tree is finite.

Theorem 7 (Computability)
Let an ITCPN be given such that $m \in \mathbb{N}$ and:

$$\forall_{t \in T} \ \forall_{c \in dom(F_t)} \ \#F_t(c) < m$$

If this ITCPN is progressive for an initial state $s \in \overline{S}$, having a finite number of tokens (i.e. $\exists_{l \in \mathbb{N}}\ \#s = l$), then the number of (really different) states reachable from s, having a minimal transition time smaller than some $y \in TS \setminus \{\infty\}$, is finite, i.e.

$$\#\{\mathcal{SB}(\hat{s}) \mid \hat{s} \in \overline{RS}(s)\ \wedge\ tt^{min}(\hat{s}) < y\} \quad \text{is finite}$$

Proof.
The ITCPN is progressive in $s \in \overline{S}$, i.e. for all $y \in TS \setminus \{\infty\}$:

$$\forall_{\sigma \in \overline{\Pi}(s)}\ \exists_{i \in dom(\sigma)}\ tt^{min}(\sigma_i) > y$$

Hence, there exists an $n \in \mathbb{N}$ such that for all $\sigma \in \overline{\Pi}(s)$:

$$\#\{i \in dom(\sigma) \mid tt^{min}(\sigma_i) \leq y\}\ \leq\ n$$

For any $\hat{s} \in \overline{S}$ having a finite number of tokens ($\#\hat{s} = l$):

(i) $\#\mathcal{SB}(\overline{R}(\hat{s}))$ is finite, because the number of really different events (disregard token identifications) allowed in $\hat{s}$ is finite (observe condition (3.4a) on page 84). In fact $\#\mathcal{SB}(\overline{R}(\hat{s})) \leq 2^l$, because 2^l is the number of possible subsets of $\hat{s}$.

(ii) For all $\hat{\hat{s}} \in \overline{R}(\hat{s}) : \#\hat{\hat{s}}$ is finite, because the number of produced tokens is smaller than m (i.e. $\#\hat{\hat{s}} < l + m$).

This implies that for all $i \in \mathbb{N}$ with $i \leq n$: $\#\mathcal{SB}(\overline{R}^i(\hat{s}))$ is finite (use induction).
This and the progressiveness property imply that the number of (really different) states reachable from s having a minimal transition time smaller than y, is finite.
□

This theorem says that the relevant part of the reachability graph, i.e. those states which have a transition time smaller than some arbitrary y, is finite. Note that we leave equivalent states aside, i.e. two states in the reachability graph, say s_1 and s_2, are considered to be equivalent if and only if $\mathcal{SB}(s_1) = \mathcal{SB}(s_2)$ (see definition 13).
To prove the property stated in theorem 7, we have to assume that: (1) the net is progressive for an initial state s, (2) s is 'finite' and (3) the number of produced tokens is always finite. This is not a surprise, since our model has a computable power equivalent to Turing machines (Wilf [127]). References [74], [93], [100] and [99] show that any significant strengthening of the basic Petri net model leads to equivalence with Turing machines. Furthermore, in [74], Jones, Landweber and Lien prove that reachability and boundedness properties are undecidable for Merlin's times Petri net model. This also holds for our ITCPN model. However, because of the three assumptions and the fact that we are only interested in the behaviour of the system until time y, these properties become decidable.
Note, the assumptions we make are not very restrictive, e.g. progressiveness is often a desirable property rather than a restriction. Recall that it is possible to

recognise the progressiveness of many nets by observing the definition of the net only (see theorem 2). Note that it is possible to adapt theorem 2 such that it holds for the modified transition system. Theorem 7 implies that if we are interested in performance measures like $\mathcal{EAT}_n$, $\mathcal{LAT}_n$, $\mathcal{LOR}$ and $\mathcal{HOR}$ or properties like K-boundedness *until* some arbitrary time y, then the MTSRT method will terminate, because the number of states to be generated is finite. Although we are able to compute upper and lower bounds for these performance measures, in some cases the time and space complexity of the algorithm may be exorbitant. This problem will be addressed in the remaining sections of this chapter.

A possible drawback of the analysis method MTSRT is the fact that answers are not always as strict as possible, because of dependencies between tokens. For example, the bounds generated for the arrival times do not have to be as tight as possible. However, experimentation shows that the calculated bounds are often of great value and far from trivial.

To our knowledge, only one analysis method has been proposed for Petri nets with interval timing. This method was presented in [17] and [16] by Berthomieu et al. and uses Merlin's timed Petri nets ([89]) to describe the system. This method also generates a reachability graph where nodes represent state classes instead of states. This approach is more or less related to our MTSRT method, although they use totally different mathematical techniques. Instead of trying to relate two transition systems, they solve linear equations to calculate state classes.
Because the method of Berthomieu et al. is based on Merlin's timed Petri net model, there are some serious drawbacks. First of all, the model does not allow for coloured tokens. This implies that it is difficult to make manageable models for large and complex systems. Secondly, they use a relative time scale, which prohibits the calculation of performance measures such as $\mathcal{EAT}_n(s,p)$ and $\mathcal{LAT}_n(s,p)$. Furthermore, it is not possible to define liveness properties such as progressiveness.
The number of states generated by Berthomieu's method is smaller than the number of states generated by the MTSRT method. However, the time needed to calculate one state is much larger. Therefore, it is difficult to determine which of these methods is most efficient, because it highly depends on the net and the initial state.
We think it is possible to extend Berthomieu's method for our ITCPN model. However, if this method uses an absolute time scale, then the computational efficiency decreases, because the number of states generated becomes comparable to the number of states generated by the MTSRT method.

3.4 Method PNRT

The MTSRT method presented in the previous section is a very powerful method, since it can be used to analyse any ITCPN. Recall, the basic idea behind this method is to construct a tree which contains at least one node for each reachable state and an arc for each possible change of state. Obviously such a tree may, even for a small

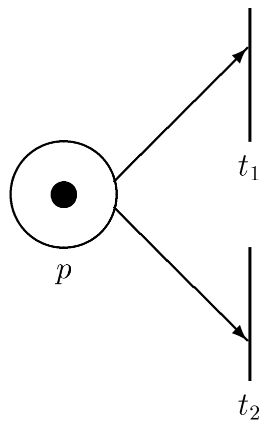


Figure 3.13: Confusion caused by a conflict between two transitions

ITCPN, become very large (and perhaps infinite). To improve the computational efficiency of this method, we want to construct a reduced reachability tree without loosing too much information. In the previous section a powerful reduction was obtained by associating time intervals with tokens rather than timestamps.

A very simple way to reduce the reachability tree is to construct the tree such that equivalent states correspond to only one node in the tree. In this case we speak about the *reachability graph* rather than the reachability tree. If there are several ways (firing sequences) to reach a specific state, this reduction is quite useful. Several authors have developed techniques to reduce the reachability graph (see Hubner et al. [67], Valmari [120], Chiola et al. [30] and Jensen [71]). These reductions often have side-effects like loosing the ability to answer certain questions. For the moment, it is only possible to construct reachability graphs for relatively small systems or parts of systems. Applying this kind of analysis to larger systems often results in an 'explosion' of the reachability graph.

Based on practical experience we identify two main causes for this 'explosion': *colour* and *confusion*.
The fact that we use coloured tokens allows us to specify a number of attributes of the entity represented by a token. Often the number of possible colours (token values) is infinite, this may result in an explosion of the reachability graph. In the next section we will concentrate on this problem.
Another phenomenon which may cause an 'explosion' of the reachability graph is called *confusion*. There are two typical forms of confusion: conflicts between transitions and conflicts between tokens.
A conflict between transitions occurs if there is a place p such that $\#(p\bullet) > 1$. Consider for example the net shown in figure 3.13. Every time there is a token in place p a non-deterministic choice has to be made: either t_1 fires or t_2 fires. In this situation there is a conflict between t_1 and t_2. If such a conflict occurs several times,

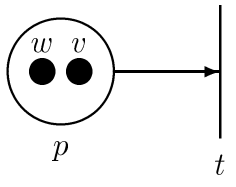


Figure 3.14: Confusion caused by a conflict between two tokens

the reachability graph is likely to 'explode'.

The second form of confusion is a conflict between two or more tokens. In the modified transition system tokens (having an identical value) are consumed in non-descending order, i.e. if tokens have equal or incomparable time intervals, a non-deterministic choice has to be made. Consider for example the situation shown in figure 3.14, where place p contains two tokens one with interval v and one with interval w. If $v = w$ and the values of the tokens differ, then there are two events possible. If $\neg(v <_i w)$, $\neg(w <_i v)$ and $v \neq w$, then the intervals are incomparable and there are also two events possible. In both cases, we say that there is a conflict between these tokens. There is no confusion if $v <_i w$ (or $w <_i v$) and the values of the tokens are identical, because in this case t consumes the token with time interval v (w).

Confusion is closely related to *persistence*. Informally speaking, an ITCPN is called persistent if, for any 'enabled' event e, the execution of another event will not 'disable' event e. An event, once it is 'enabled', will stay enabled until it occurs. Clearly, any form of confusion endangers persistence. However, the absence of confusion does not guarantee persistence. To guarantee persistence of an ITCPN with respect to some initial state, we have to add the requirement that the time intervals of the tokens in each place have to be ascending in order of arrival, i.e. tokens produced for a place have a time interval of at least the time interval of any token already present in this place. Consider the ITCPN shown in figure 3.15. Initially, there is one token in p_1 with interval v, there is one token in p_2 with interval w and there is one token in p_3 with interval u. Suppose $v = \langle 0, 4\rangle$, $w = \langle 2, 6\rangle$ and $u = \langle 0, 0\rangle$. Initially, two events are enabled. Event e_1 corresponds to the firing of t_1 ($et^{min}(e_1) = 0$, $et^{max}(e_1) = 4$) and event e_2 corresponds to the firing of t_2 ($et^{min}(e_1) = 2$, $et^{max}(e_1) = 6$). If e_1 occurs first, then e_2 may become disabled, because $v <_i w$ and the tokens are consumed in non-descending order (provided that they have the same value), i.e. t_2 may consume the token produced by t_1 rather than the token with interval w. Nevertheless, there is no confusion. This example shows that the absence of confusion does not imply persistence.

To formalize the persistence concept, we start with the definition of a *well-ordered* state.

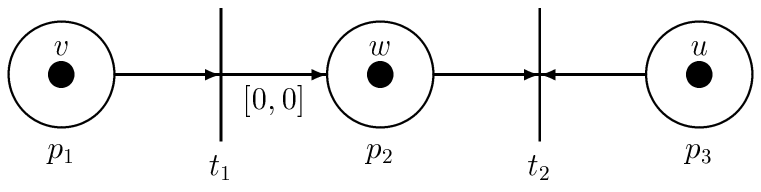


Figure 3.15: A non-persistent ITCPN without confusion

Definition 32 (Well-ordered)
A state $s \in \overline{S}$ is *well-ordered*, if and only if, for any $i, j \in dom(s)$:

$$place(s(i)) = place(s(j)) \Rightarrow (time(s(i)) \leq_i time(s(j)) \ \vee \ time(s(j)) \leq_i time(s(i)))$$

A state is well-ordered if the time intervals of any pair of tokens in the same place are comparable. In other words, of any two tokens in the same place, one interval is smaller than or equal to the other.

Definition 33 (Persistence)
An ITCPN is *persistent* with respect to $s \in \overline{S}$, if and only if:

1. the net is conflict free
2. for any $\hat{s} \in \overline{RS}(s)$: $\hat{s}$ is well-ordered
3. for any $\hat{s} \in \overline{RS}(s)$, $\tilde{s} \in \overline{R}(\hat{s})$, $i \in dom(\hat{s})$ and $j \in dom(\tilde{s}) \setminus dom(\hat{s})$:

$$place(\hat{s}(i)) = place(\tilde{s}(j)) \Rightarrow time(\hat{s}(i)) \leq_i time(\tilde{s}(j))$$

The third requirement says that the time intervals of the tokens arriving in each place have to be ascending in the order of their arrival. All produced tokens have a time interval of at least any interval of the tokens contained by the (corresponding) place until then.
A persistent net has the nice property that, if an event ‘occurs’, then it will not ‘disable’ any other event (this will be formalized later).
Note that our definition of persistence slightly deviates from the more traditional definition, where persistence means that the firing of a transition will not disable any other enabled transition (see Murata [93]).

In this section we concentrate on persistent nets. Clearly, for an arbitrary net (and initial state) it may be difficult to verify whether the net is persistent. Therefore, we will show that an important class of ITCPNs is persistent. We will use an example to demonstrate that this class allows for the modelling of meaningful repetitive manufacturing processes.

Persistent interval timed coloured Petri nets have a number of interesting properties. We have developed an analysis method that exploits persistence to reduce the reachability graph. The method is called *Persistent Net Reduction Technique* (PNRT). This technique is based on a slightly altered version of the modified transition system used by the MTSRT method. In this section we restrict ourselves to 'uncoloured' Petri nets. The extension of this method to coloured nets is straightforward if we add some additional requirements (inter alia the requirement that no two tokens in a place have the same time interval).

Assumption
The interval timed coloured Petri nets considered in this section are 'colourless' (i.e. $\forall_{p \in P} \ \#V_p = 1$) and the corresponding modified transition system is altered in the following way, equation (3.10) is replaced by (3.10′):

$$s_1 \overline{R} s_2 \quad \equiv \quad \exists_{\substack{e \in \overline{AE}(s_1) \\ et^{min}(e) \leq tt^{max}(s_1)}} \quad s_2 = (s_1 \setminus \pi_2(e)) \ \cup \ \overline{scale}(\pi_3(e), et^{min}(e), et^{max}(e)) \quad \text{(3.10')}$$

Note that $tt^{max}(s_1)$ is replaced by $et^{max}(e)$. This assumption is valid for the rest of section 3.4.

We assume a colourless ITCPN to avoid confusion between tokens having the same time interval. Replacing (3.10) by (3.10′) makes the timestamps of the produced tokens independent of the other (allowed) events. Since $et^{max}(e) \geq tt^{max}(s_1)$, many of the properties mentioned in the previous section remain valid, e.g. the soundness property. Moreover, performance measures, such as $\mathcal{EAT}_n$, $\mathcal{LAT}_n$, $\mathcal{HOR}$ and $\mathcal{LOR}$, calculated using this transition system are still 'safe'.

A persistent ITCPN with respect to s has the nice property that, if it is dead w.r.t. s, then it always terminates in the 'same' state. This property is stated in the following theorem:

Theorem 8
If an ITCPN is persistent and dead with respect to an initial state $s \in \overline{S}$, then:

$$\#\{\mathcal{SB}(\hat{s}) \mid \hat{s} \in \overline{RS}(s) \ \land \ \overline{R}(\hat{s}) = \emptyset\} = 1$$

Proof.
Because the cardinality of the colour set of each place is 1 (i.e. $\forall_{p \in P} \ \#V_p = 1$) and all $\hat{s} \in \overline{RS}(s)$ are well-ordered (see definition of persistence), there are no conflicts between tokens, i.e. if there are two tokens in a place p with time intervals v and w, then $v \leq_i w$ or $w \leq_i v$. Note that if $v = w$, then the tokens are identical, although they may have different labels.
Two events are equivalent if the consumed tokens are identical w.r.t. their time interval (and value). More formally: for any $e_1, e_2 \in \overline{E}$, we have: $e_1 \doteq e_2$, if and only if, $\pi_1(e_1) = \pi_1(e_2)$ and $\mathcal{SB}(\pi_2(e_1)) = \mathcal{SB}(\pi_2(e_2))$ and $\mathcal{SB}(\pi_3(e_1)) = \mathcal{SB}(\pi_3(e_2))$. Since there are no conflicts between tokens, $e_1, e_2 \in \overline{AE}(\hat{s})$ and $\pi_1(e_1) = \pi_1(e_2)$ imply that $e_1 \doteq e_2$.
Once an event e is 'enabled', i.e. $e \in \overline{AE}(\hat{s})$, it remains enabled until it occurs. In other words, an event can and will not be disabled by any other event. If another event, say h ($h \neq e$), occurs in $\hat{s}$, then e is still enabled in: $\hat{\hat{s}} = (\hat{s} \setminus \pi_2(h)) \cup \overline{scale}(\pi_3(h), et^{min}(h), et^{max}(h))$, because:

1. $dom(\pi_2(h)) \cap dom(\pi_2(e)) = \emptyset$, because of the absence of conflicts between transitions and $\pi_1(h) \neq \pi_1(e)$. Consequently, $\pi_2(e) \subseteq \hat{\hat{s}}$, i.e. condition (3.4a) in the definition of $\overline{AE}$ holds (see page 84).

2. For any $i \in dom(\pi_2(e))$ and $j \in dom(\hat{\hat{s}}) \setminus dom(\pi_2(e))$, we have:
$place(\hat{\hat{s}}(i)) = place(\hat{\hat{s}}(j)) \Rightarrow \neg(time(\hat{\hat{s}}(j)) <_i time(\hat{\hat{s}}(i)))$,
because the produced tokens have time intervals which are *not* smaller than the tokens already present in the corresponding place (persistence). Therefore, condition (3.4c) in the definition of $\overline{AE}$ holds.

3. The other conditions (3.4b, 3.4d and 3.4e) in the definition of $\overline{AE}$ still hold for e (sometimes $\pi_3(e)$ has to be relabelled, because some of its labels are already used).

If an event e occurs, the intervals of the produced tokens only depend upon e and not upon any other event (see equation (3.10′)). This implies that the ordering of events is not important, i.e. all firing sequences executing a given set of events result in the 'same' state.
Moreover, if $et^{min}(e) \leq tt^{max}(\hat{s})$, then $et^{min}(e) \leq tt^{max}(\hat{\hat{s}})$, because tt^{max} is ascending (see theorem 5).
The net is dead, therefore the set of enabled events becomes empty after a while. This and the fact that an event will not be disabled implies that all firing sequences executing a given set of events result in the 'same' state, i.e. $\#\{\mathcal{SB}(\hat{s}) \mid \hat{s} \in \overline{RS}(s) \ \land \ \overline{R}(\hat{s}) = \emptyset\} = 1$. Suppose that this is *not* the case (i.e. there are multiple terminal states), then there are two paths σ and σ' resulting in a different terminal state. We just showed that all firing sequences executing a given set of events

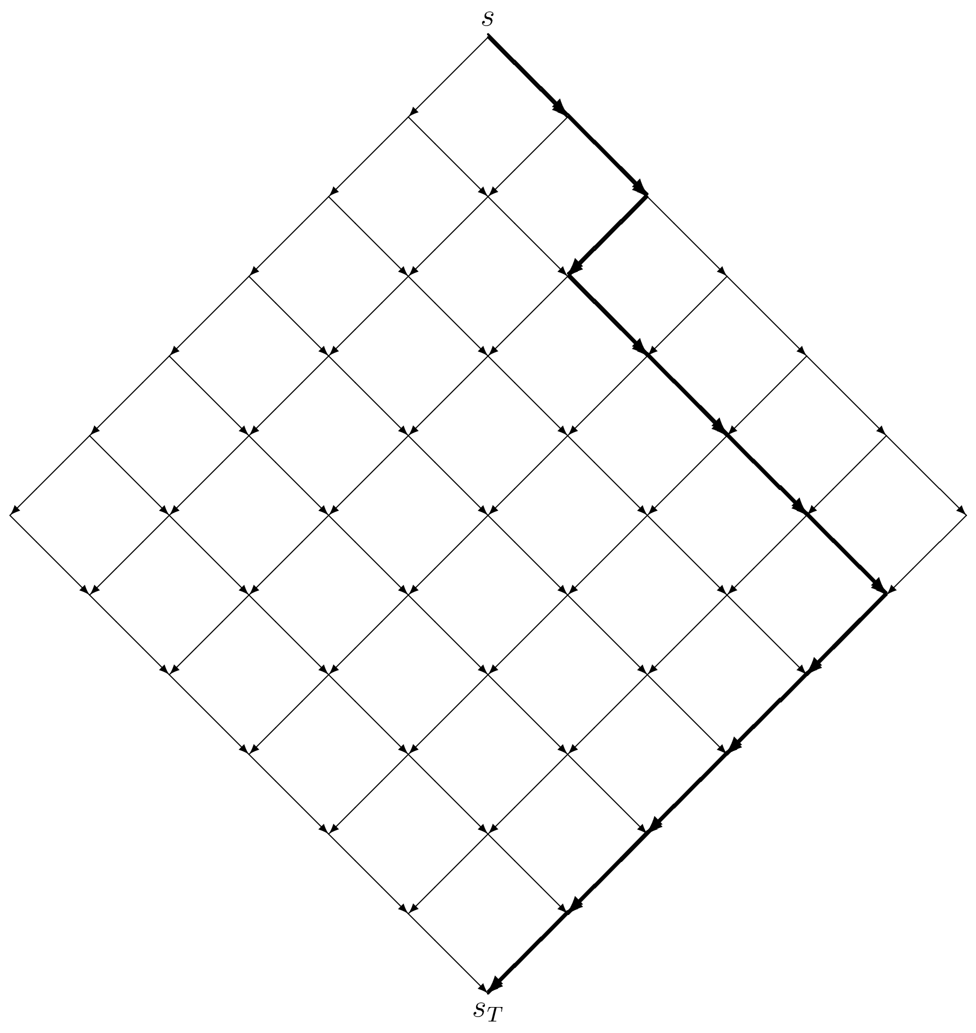


Figure 3.16: The reachability graph of a persistent dead ITCPN

result in the 'same' state. Hence, there is an event e_k transforming σ_{k-1} into σ_k ($k \in dom(\sigma) \setminus \{0\}$), which does not 'occur' in the firing sequence σ'. By backtracking these firing sequences we learn that this is not possible. We do not prove this formally, but rely on the intuition of the reader (see figure 3.16).
□

This theorem tells us that it does not matter which events are chosen during the execution of the net, i.e. all paths (firing sequences) lead to the same terminal state in the modified transition system. Therefore, this terminal state can be calculated very efficiently, i.e. resolve all choices by selecting an arbitrary event. Figure 3.16 illustrates this property.

For an arbitrary net it is very difficult to verify whether the net is persistent. However, there is an important class of nets for which we can prove that they are persistent. This is expressed by theorem 9. To prove theorem 9, we need the following lemma which tells us that the maximal (interval) sequence of two ascending (interval) sequences is ascending.

Lemma 17
If $n \in \mathbb{N}$, $v_1, v_2, .., v_n \in INT$ and $w_1, w_2, .., w_n \in INT$ such that $\forall_{i \in \{1..n-1\}}\ (v_i \leq_i v_{i+1})\ \wedge\ (w_i \leq_i w_{i+1})$, then: [3]

$$\forall_{i \in \{1..n-1\}}\ (v_i \max w_i) \leq_i (v_{i+1} \max w_{i+1})$$

Proof.
For $i \in \{1..n-1\}$, $v_i \leq_i v_{i+1}\ \wedge\ w_i \leq_i w_{i+1}$ implies that
$\pi_1(v_i) \leq \pi_1(v_{i+1})$, $\pi_1(w_i) \leq \pi_1(w_{i+1})$, $\pi_2(v_i) \leq \pi_2(v_{i+1})$ and $\pi_2(w_i) \leq \pi_2(w_{i+1})$.
$\pi_1(v_i \max w_i) = \pi_1(v_i) \max \pi_1(w_i) \leq \pi_1(v_{i+1}) \max \pi_1(w_{i+1}) = \pi_1(v_{i+1} \max w_{i+1})$
$\pi_2(v_i \max w_i) = \pi_2(v_i) \max \pi_2(w_i) \leq \pi_2(v_{i+1}) \max \pi_2(w_{i+1}) = \pi_2(v_{i+1} \max w_{i+1})$
Therefore: $(v_i \max w_i) \leq_i (v_{i+1} \max w_{i+1})$.
$\square$

In chapter 2 we defined a *marked graph* as follows: a marked graph (or timed event graph) is an ordinary ITCPN such that each place has 0 or 1 input transitions and 0 or 1 output transitions, i.e. $\forall_{p \in P}\ \#(\bullet p) \leq 1\ \wedge\ \#(p\bullet) \leq 1$.
Recall, a *source place*, is a place without any input transitions, i.e. $P^S = \{p \in P \mid \bullet p = \emptyset\}$ is the set of source places.
A marked graph is persistent, if the initial state is well-ordered, all tokens in the 'non-source' places $(P \setminus P^S)$ have the same interval, say v, and every token in a source place has a time interval of at least v (i.e. $\geq_i v$). This property of marked graphs is expressed in the following theorem.

Theorem 9
A marked graph with an initial state $s \in \overline{S}$ such that:

1. s is well-ordered
2. $\forall_{i,j \in dom(s)}\ (place(s(i)) = place(s(j)) \in (P \setminus P^S)) \Rightarrow\ time(s(i)) = time(s(j))$
3. $\forall_{i,j \in dom(s)}\ place(s(i)) \in (P \setminus P^S)\ \wedge\ place(s(j)) \in P^S \Rightarrow$
   $time(s(i)) \leq_i time(s(j))$

is persistent with respect to s.

Proof.
By definition a marked graph is conflict free. Remains to prove that:

(i) for any $\hat{s} \in \overline{RS}(s)$: $\hat{s}$ is well-ordered

(ii) for any $\hat{s} \in \overline{RS}(s)$ and $\tilde{s} \in \overline{R}(\hat{s})$, $i \in dom(\hat{s})$ and $j \in dom(\tilde{s}) \setminus dom(\hat{s})$:
$place(\hat{s}(i)) = place(\tilde{s}(j)) \Rightarrow time(\hat{s}(i)) \leq_i time(\tilde{s}(j))$

[3]If $v, w \in INT$, then $v \max w = \langle \pi_1(v) \max \pi_1(w), \pi_2(v) \max \pi_2(w) \rangle$.

Suppose that (ii) holds, in this case it is easy to prove (i). If $\hat{s} \in \overline{RS}(s)$, then there exists an $n \in \mathbb{N}$ such that $\hat{s} \in \overline{R}^n(s)$. Let $P(n)$ be the proposition that all $\hat{s} \in \overline{R}^n(s)$ are well-ordered. $P(0)$ is trivial, because $\hat{s} \in \overline{R}^0(s) = \{s\}$ is well-ordered. Suppose $n > 0$ and $P(n-1)$ (induction hypothesis). For all $\tilde{s} \in \overline{R}^n(s)$ there exists a state $\hat{s} \in \overline{R}^{n-1}(s)$ such that $\tilde{s} \in \overline{R}(\hat{s})$. Because $\hat{s}$ is well-ordered (induction), the corresponding event e which transforms $\hat{s}$ into $\tilde{s}$ adds *one* token to each output place such that the state remains well-ordered. This is guaranteed by the fact that the net is a marked graph and for any produced token with interval v and any token (with interval w) contained by the corresponding place until then, we have $w \leq_i v$ (see (ii)).

Remains to prove that (ii) holds. For convenience, we define $Q(p)$ as follows:

$$\begin{aligned} Q(p) \quad \equiv \quad & \forall_{\hat{s} \in \overline{RS}(s)} \forall_{\tilde{s} \in \overline{R}(\hat{s})} \forall_{i \in dom(\hat{s})} \forall_{j \in dom(\tilde{s}) \setminus dom(\hat{s})} \\ & (place(\hat{s}(i)) = p \ \wedge \ place(\tilde{s}(j)) = p) \ \Rightarrow \ time(\hat{s}(i)) \leq_i time(\tilde{s}(j)) \end{aligned}$$

Note that $\forall_{p \in P} \ Q(p)$ implies (ii).
A first observation tells us that $Q(p)$ holds for all tokens in the source places P^S, because no event will add tokens to one of these places.
If $t \in T$ is a transition such that the tokens in each of its input places satisfy requirement (ii) (i.e. for all $p \in \bullet t$: $Q(p)$), then each output place also satisfies (ii) (i.e. for all $p \in t\bullet$: $Q(p)$), because t is the only transition producing tokens for these places, the tokens initially available satisfy (1.), (2.) and (3.) and lemma 17 tells us that if the intervals of the tokens on the input places are ascending, then the tokens in the output places are also ascending.

Consider a place $p \in P$ with $\bullet p \neq \emptyset$. Suppose that $Q(p)$ does not hold, then there exists a state $\hat{s} \in \overline{RS}(s)$ with a token in p with time interval v and an event e which transforms $\hat{s}$ into $\tilde{s}$, such that e adds tokens to p with an interval w which is not at least v, i.e. $\neg(v \leq_i w)$.
In this case, either the token with time interval v already existed in the initial state s or the token with time interval v was produced by the same transition t which produced the token with time interval w.
If the token already existed in s, then $v \leq_i w$ (i.e. a contradiction), because all tokens produced by some transition have an interval of at least v (see requirements (2.) and (3.)). Hence, both tokens have been produced by the same transition t (every place has only one input transition). But this means that one of the input places of t contained a token with interval $\hat{v}$ and a token with interval $\hat{w}$ such that the token with interval $\hat{v}$ existed before the token with interval $\hat{w}$ and $\neg(\hat{v} \leq_i \hat{w})$, this follows from lemma 17. Continue this reasoning until a contradiction is encountered, either because all input places of t have no incoming arcs or because one reaches the initial state s which is well-ordered.
Hence, $Q(p)$ holds for any place p.
$\square$

This theorem tells us that, given some conditions, a marked graph is persistent. If the net is dead, then there is only one terminal state in the modified transition system. This terminal state can be calculated very efficiently. The time complexity of the PNRT method is $\mathcal{O}(\#\sigma(\#P + \#T))$, where σ is an arbitrary execution path (the time required to calculate an event and to execute this event is $\mathcal{O}(\#P+\#T)$, see Van den Heuvel [61]). Note that this is comparable to the time needed to simulate the net once, i.e. one simulation run of length $\#\sigma$ (events).

Since the soundness properties stated in section 3.3 are also valid for the transition system used by the PNRT method, we can answer a number of questions. For example, we can calculate the earliest n^{th} arrival time ($\mathcal{EAT}_n$) and the latest n^{th} arrival time ($\mathcal{LAT}_n$) of sink places, i.e. places without outgoing arcs. Note that these bounds are as 'tight' as possible.

The dynamic behaviour of (timed) marked graphs (timed event graphs) has been studied by a lot of people. Analysis techniques to analyse the steady-state behaviour of a marked graph have been presented by Ramamoorthy and Ho in [107] and Chretienne et al. in [28] and [31]. These authors analyse timed marked graphs where a deterministic delay is associated with each transition in the net. These analysis techniques evaluate all circuits to calculate the 'performance' of the system.

A generalization of these methods has been presented by Van der Aalst in section 5 of [2]. The method described in this report is called the *Steady State Performance Analysis Technique* (SSPAT). It is a generalization in the sense that it is based on the ITCPN model which uses interval delays rather than deterministic delays. The SSPAT method calculates upper and lower bounds for the 'performance' of the system. A detailed description of this method is not included in this monograph, because it can only be applied to strongly connected marked graphs (i.e. periodically operated Petri nets) and it does not answer any of the performance measures defined in chapter 2 (this also holds for the other techniques described in [107], [28] and [31]).

A lot of applications have been modelled and analysed using marked graphs, see for example Hillion and Proth [62], Silva and Valette [115] or Chretienne et al. [28]. Typical application areas of timed marked graphs are: project engineering (see section 3.2.1), flexible manufacturing and production scheduling. To illustrate the modelling power of timed marked graphs, we model a small production system in terms of a marked graph that will be analysed using the PNRT method.

The production system we are interested in, produces items named $\mathcal{H}$ using raw materials $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. There are also a number of intermediate products: $\mathcal{D}$, $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$. There are three machines, **M1** transforms $\mathcal{A}$ into $\mathcal{D}$, **M2** transforms $\mathcal{B}$ into $\mathcal{E}$ and **M3** transforms $\mathcal{C}$ into $\mathcal{F}$. There is one subassembly composing $\mathcal{D}$ and $\mathcal{E}$ into $\mathcal{G}$ and one final assembly composing $\mathcal{G}$ and $\mathcal{F}$ into $\mathcal{H}$. Figure 3.17 shows the bill of materials.

The ITCPN shown in figure 3.18 is used to model the production system. Places

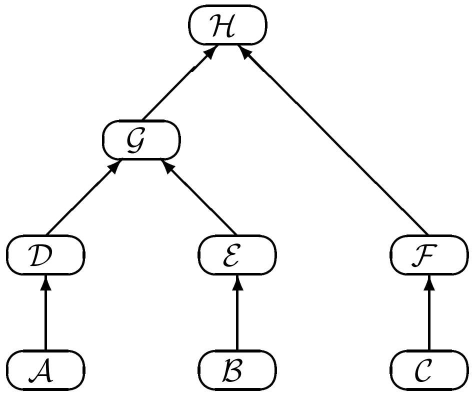


Figure 3.17: The bill of materials

p1,*p2*, .. and *p11* are used to represent the flow of products. Raw materials $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ enter the system via places *p1*, *p2* and *p3* respectively. Product $\mathcal{D}$ is stored in *p6*, $\mathcal{E}$ in *p7*, $\mathcal{F}$ in *p8*, $\mathcal{G}$ in *p9* and $\mathcal{H}$ in *p10*. Finished products $\mathcal{H}$ leave the system via place *p11*. The demand for product $\mathcal{H}$ arrives via the place *demand*.

Note that we use the initial state to represent the behaviour of the environment (e.g. demand and supply). In this way we can analyse the system under various circumstances, without changing the net (see 2.6).

Machine **M3** transforms products $\mathcal{C}$ into $\mathcal{F}$ and is modelled by a queueing system represented by the subnetwork containing transitions *t1* and *t2*. Initially, there is one token in place *free3* indicating that the machine is ready to operate.

Machines **M1** and **M2** need a setup every time an item is processed. This setup is performed by a person working on both machines. We may think of this person as a *shared resource*. The setup of **M1** is represented by transition *t4*, the setup of **M2** is represented by transition *t3*. The person is represented by a token in place *h1* or place *h2*. Note that the person alternates between **M1** and **M2**. The remaining parts of **M1** and **M2** are modelled similar to **M3**. Note that we use a *push* control to direct machines **M1**, **M2** and **M3**. Each time raw material is available and the machine is free, an operation is started.

We use a *pull* control to direct the two assembly processes (i.e. assemble to order). In this example a *Kanban*-like control technique is used to reduce the in-process inventory. This technique has been developed in Japan to achieve a Just-in-Time production (see Sugimori et al. [117]). Assembling is allowed if the components needed for the assembly are available and if a certain *card*, called Kanban, has been received. A new Kanban is supplied the moment an assembled product is removed. In this way one gets a demand-driven assembly process.

The subassembly and the final assembly are represented by *t9* and *t10*. The delivery of item $\mathcal{H}$ is modelled by transition *t11*. Transition *t11* fires, if there is a demand

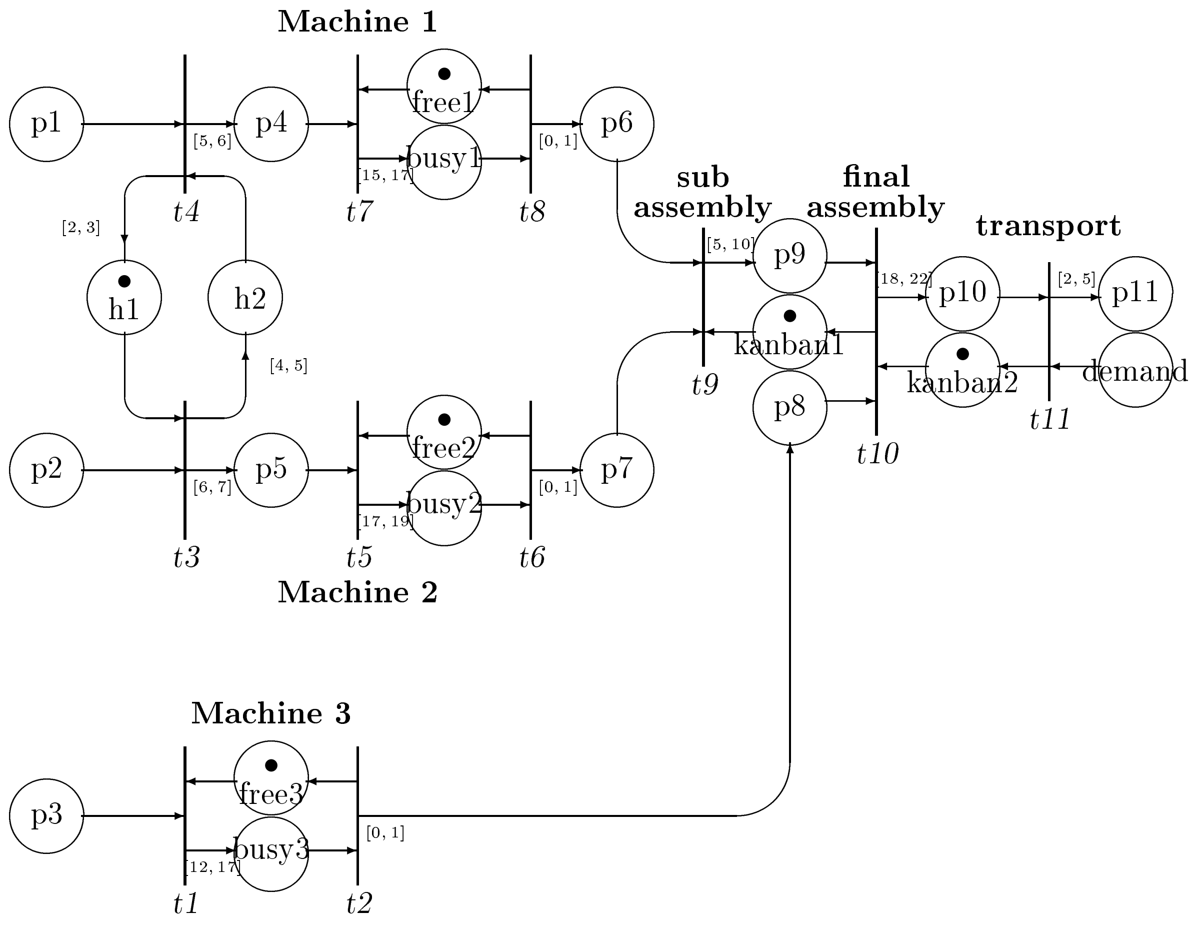


Figure 3.18: A production system

and a finished product. If *t11* fires, a new Kanban is supplied to the final assembly process (*t10*). If *t10* fires, a new Kanban is supplied to the subassembly process (*t9*). Note that the maximum amount of stored products $\mathcal{G}$ and $\mathcal{H}$ depends on the number of tokens initially available in *kanban1* and *kanban2*.
Figure 3.18 also shows the delay intervals associated with every time consuming operation.

Let us assume that the production system receives a steady flow of raw materials ($\mathcal{A}$,$\mathcal{B}$ and $\mathcal{C}$). Every 20 minutes the system receives an order for one product $\mathcal{H}$ (starting at time 0). Initially, there is one Kanban in *kanban1* and one Kanban in *kanban2*. Now we are interested in the arrival times of tokens in place *p11*. Table 3.1

ordernumber (n)	$\mathcal{EAT}_n$	$\mathcal{LAT}_n$	minimal lead time	maximal lead time
1	49	66	49	66
2	69	88	49	68
3	89	110	49	70
10	229	264	49	84
50	1029	1144	49	164

Table 3.1: Some results obtained using the PNRT method

shows some results obtained using method PNRT. Note that this is possible because all the conditions of theorem 9 are satisfied, i.e. the net is persistent. For example the 10^{th} order (generated after (10-1)*20 = 180 minutes) was delivered between 229 ($\mathcal{EAT}_{10}$) and 264 ($\mathcal{LAT}_{10}$) minutes. Therefore, the lead time of this order is between 49 and 84 minutes.
The maximal lead time is increasing, because the final assembly of product $\mathcal{H}$ may need 22 minutes and this is longer than the interarrival time (=20 minutes). The minimal lead time is constant, because under ideal circumstances there is an abundance of capacity.

The PNRT analysis method calculates the terminal state of a marked graph very efficiently. There are however some drawbacks. First of all, there is the limitation that the PNRT method can only be applied to marked graphs or, more precisely, persistent nets. Another restriction is the fact that the method only obtains results about the terminal state, therefore it is not possible to calculate performance measures like $\mathcal{LOR}$ and $\mathcal{HOR}$. Thirdly, the net has to be dead. This is not a serious restriction, because we are often interested in nets with a number of source places representing the input of the system and these nets are usually dead, i.e. if we use a finite initial state s to model the environment, then the net is often dead w.r.t. s.
If we want to analyse nets that are not dead, then we can use the SSPAT method described in [2] to analyse the steady-state performance of the net. Finally, there is the restriction that the PNRT method described in this section cannot be applied to 'coloured' nets, i.e. $\forall_{p \in P}\ \#V_p = 1$. To relax this restriction, we have to impose other ones.
Note that if one of these limitations prevents us from using the PNRT method, we can always resort to the MTSRT method described in the previous section.

3.5 Dealing with large colour sets

The MTSRT method presented in section 3.3 is a very powerful analysis method, since it can be applied to almost any ITCPN encountered in practice. An obvious restriction of this method is that the reduced reachability graph constructed by

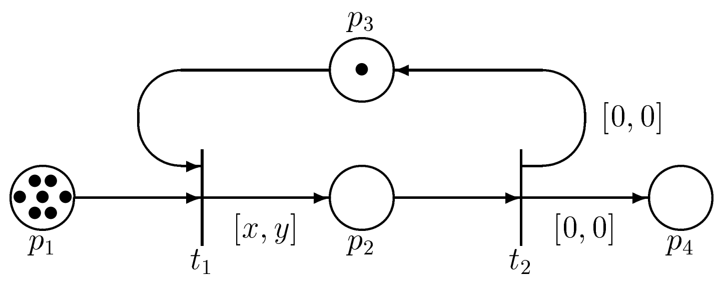


Figure 3.19: A queueing system modelled by an ITCPN

the MTSRT method may become very large, thus making analysis time and space consuming. We already mentioned the two main causes for such an explosion: *colour* and *confusion*. In the previous section we saw that, if we are able to avoid confusion (e.g. by using marked graphs), then we can use more efficient methods like the PNRT method. In this section we demonstrate techniques to deal with computational problems caused by the colouring of tokens.

Consider the following ITCPN:

$P = \{p_1, p_2, p_3, p_4\}$
$V_{p_1} = \mathbb{N}$, $V_{p_2} = \mathbb{N}$, $V_{p_3} = \{\emptyset\}$ and $V_{p_4} = \mathbb{N}$
$T = \{t_1, t_2\}$
$I = \{\langle t_1, [p_1, p_3]\rangle, \langle t_2, [p_2]\rangle\}$
$O = \{\langle t_1, \{p_2\}\rangle, \langle t_2, \{p_3, p_4\}\rangle\}$
For all $k \in \mathbb{N}$:
$F_{t_1}([\langle p_1, k\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, k\rangle, \langle (k \bmod 3) + 10, (k \bmod 3) + 15\rangle\rangle]$
$F_{t_2}([\langle p_2, k\rangle]) = [\langle\langle p_3, \emptyset\rangle, \langle 0, 0\rangle\rangle, \langle\langle p_4, k\rangle, \langle 0, 0\rangle\rangle]$

Figure 3.19 shows the graphical representation of this ITCPN. Initially, there is one token in place p_3 with value $\emptyset$ and timestamp 0. There are n tokens in place p_1 also with timestamp 0 and the corresponding values range from 1 to n, i.e. there is one token with value 1, one token with value 2, .. etc. The reachability tree used by the MTSRT method contains $n!$ different terminal states. If $n = 50$ the MTSRT method has to evaluate $3.04 \cdot 10^{64}$ different firing sequences of length 100 to calculate $\mathcal{EAT}_{50}(s, p_4) = 51 + 50 \cdot 10 = 551$ and $\mathcal{LAT}_{50}(s, p_4) = 51 + 75 \cdot 10 = 801$. This explosion of the reduced reachability tree is caused by the fact that the tokens in p_1 (p_2,p_4) have different values. If t_1 fires for the first time, it has to make a non-deterministic choice of 50 tokens all having a different value. If t_1 fires for the second time, it has to make a non-deterministic choice of 49 tokens, etc. We will use this example to illustrate how to deal with these explosions caused by relatively large colour sets.

3.5.1 Approach 1: remove the colour

A straightforward but rigourous approach is to 'remove' the colouring. Removing the colouring does not affect the network structure, i.e. P, T, I and O remain the same. The value set (colour set) of each place is replaced by a set containing one element (e.g. $\emptyset$), i.e. $\forall_{p \in P}\ V_p = \{\emptyset\}$. To produce 'safe' results, F_{t_1} and F_{t_2} are modified such that the lower (upper) bound of the delay interval of a produced token corresponds to the smallest (largest) possible delay. For the example shown in figure 3.19:

$F'_{t_1}([\langle p_1, \emptyset\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, \emptyset\rangle, \langle 10, 17\rangle\rangle]$
$F'_{t_2}([\langle p_2, \emptyset\rangle]) = [\langle\langle p_3, \emptyset\rangle, \langle 0, 0\rangle\rangle, \langle\langle p_4, \emptyset\rangle, \langle 0, 0\rangle\rangle]$

Note that $\min\{(k \bmod 3) + 10 \mid k \in \mathbb{N}\} = 10$ and $\max\{(k \bmod 3) + 15 \mid k \in \mathbb{N}\} = 17$. In this case the MTSRT method calculates only one terminal state, i.e. the MTSRT method has to evaluate only one firing sequence of length 50 to calculate $\mathcal{EAT}'_{50}(s, p_4) = 500$ and $\mathcal{LAT}'_{50}(s, p_4) = 850$.
Although these bounds are not as 'tight' as possible, they are safe in the sense that: $\mathcal{EAT}'_{50}(s, p_4) \leq \mathcal{EAT}_{50}(s, p_4)$ and $\mathcal{LAT}'_{50}(s, p_4) \geq \mathcal{LAT}_{50}(s, p_4)$. We will prove that this is always the case provided that the number of produced tokens is independent of the values of the consumed tokens.

Replacing an ITCPN by a colourless ITCPN is called *uncolouring*. Uncolouring is only possible if the following assumption holds.

Assumption
There is a function $prod \in (T \times P) \rightarrow \mathbb{N}$, such that for any $t \in T$ and $p \in P$:

$$\forall_{c \in dom(F_t)} \ (\sum_{\substack{q \in F_t(c) \\ place(q) = p}} F_t(c)(q)) = prod(t, p)$$

This assumption is used throughout section 3.5.1. Informally speaking, this assumption restricts the class of nets we consider to those nets where the number of tokens produced by any transition does not depend upon the values of the consumed tokens.

Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN and let $N' = (P', V', T', I', O', F', TS')$ be the corresponding uncoloured ITCPN. First, we show how to construct this N', then we will investigate the relation between these nets.

The set of places of the uncoloured ITCPN equals the set of places of the coloured net, i.e. $P' = P$. Similar statements hold for the set of transitions, the input places, the output places and the time set, i.e. $T' = T$, $I' = I$, $O' = O$ and $TS' = TS$. The value set of each place is the set $\{\emptyset\}$, i.e. $dom(V') = P$ and for all $p \in P$: $V'_p = \{\emptyset\}$. If $t \in T$, then $dom(F'_t) = \{uncolour(c) \mid c \in dom(F_t)\}$, where $uncolour \in$

$\mathbb{B}(CT) \rightarrow \mathbb{B}(CT')$ such that for $c \in \mathbb{B}(CT)$:

$$uncolour(c) = \lambda_{\langle p,\emptyset\rangle \in CT'} \ (\sum_{v \in V_p} c(< p, v >))$$

Note that we use primes (e.g. CT') to avoid confusion between symbols corresponding to N and N'. The function *uncolour* transforms a bag of 'coloured' tokens into a bag of 'uncoloured' tokens, i.e. tokens with value $\emptyset$. Note that $\#dom(F'_t) = 1$, because for any $c \in dom(F_t)$: $uncolour(c) = \lambda_{\langle p,\emptyset\rangle \in CT'} \ I_t(p)$.
To define F', we need to determine the smallest and largest possible delay of a token produced by a transition $t \in T$ for a place $p \in P$.

$$\begin{aligned} low(t,p) &= \min_{c \in dom(F_t)} \ \min\{time^{min}(q) \mid q \in F_t(c) \ \wedge \ place(q) = p\} \\ high(t,p) &= \max_{c \in dom(F_t)} \ \max\{time^{max}(q) \mid q \in F_t(c) \ \wedge \ place(q) = p\} \end{aligned}$$

Any token produced by a firing of transition t for a place p has a delay between $low(t,p)$ and $high(t,p)$. If p is not an output place of t, then $low(t,p) = \infty$ and $high(t,p) = -\infty$.
The delays of the tokens in N may depend upon the values of the consumed tokens. Removing the colouring implies that the delays have to become independent of the tokens consumed. Therefore, the delays in N' are sampled from a delay interval containing all the corresponding delay intervals in N. More formally, for any $t \in T$:

$$F'_t(\lambda_{\langle p,\emptyset\rangle \in CT'} \ I_t(p)) = \lambda_{\langle\langle p,\emptyset\rangle,\langle x,y\rangle\rangle \in CT' \times INT} \begin{cases} prod(t,p) & \text{if } x = low(t,p) \text{ and} \\ & \quad y = high(t,p) \\ 0 & \text{otherwise} \end{cases}$$

If we apply these rules properly, then the uncoloured ITCPN corresponding to the coloured ITCPN shown in figure 3.19 is defined as follows:

$P' = \{p_1, p_2, p_3, p_4\}$
$V'_{p_1} = V'_{p_2} = V'_{p_3} = V'_{p_4} = \{\emptyset\}$
$T' = \{t_1, t_2\}$
$I' = \{\langle t_1, [p_1, p_3]\rangle, \langle t_2, [p_2]\rangle\}$
$O' = \{\langle t_1, \{p_2\}\rangle, \langle t_2, \{p_3, p_4\}\rangle\}$
$F'_{t_1}([\langle p_1, \emptyset\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, \emptyset\rangle, \langle 10, 17\rangle\rangle]$
$F'_{t_2}([\langle p_2, \emptyset\rangle]) = [\langle\langle p_3, \emptyset\rangle, \langle 0, 0\rangle\rangle, \langle\langle p_4, \emptyset\rangle, \langle 0, 0\rangle\rangle]$

In general, the (reduced) reachability tree of the uncoloured ITCPN is much smaller than the (reduced) reachability tree of the coloured ITCPN. Obviously there is some relation between the transition systems of these two nets. We want to use the uncoloured net to answer questions about the coloured net, therefore we have to establish a formal relationship between the corresponding transition systems.
It is easy to see that the transition systems are not equivalent. There is, however, a very convenient morphism between the transition systems of N and N'.

Theorem 10
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN, the semantics of which is described by a transition system $X = \langle S, R \rangle$ and let $N' = (P', V', T', I', O', F', TS')$ be the corresponding uncoloured ITCPN, the semantics of which is described by a transition system $Y = \langle S', R' \rangle$. Then the function $rmc \in S \to S'$ is a morphism from X to Y, where rmc is defined as follows:

$$dom(rmc) = S$$
$$\forall_{s \in S}\ rmc(s) = \lambda_{i \in dom(s)}\ \langle \langle place(s(i)), \emptyset \rangle, time(s(i)) \rangle$$

Proof.
For any $s_1, s_2 \in S$ such that $s_1 R s_2$, we have to prove that $rmc(s_1) R' rmc(s_2)$. Because $s_1 R s_2$, there exists an event e such that:

(i) $e \in AE(s_1)$

(ii) $et(e) = tt(s_1)$

(iii) $s_2 = (s_1 \setminus \pi_2(e)) \cup scale(\pi_3(e), tt(s_1))$

Define $e' = \langle \pi_1(e), rmc(\pi_2(e)), rmc(\pi_3(e)) \rangle \in E'$.

Now it suffices to prove that:

(i) $e' \in AE'(rmc(s_1))$

(ii) $et(e') = tt(rmc(s_1))$

(iii) $rmc(s_2) = (rmc(s_1) \setminus \pi_2(e')) \cup scale(\pi_3(e'), tt(rmc(s_1)))$

(i) Event e' is an element of $AE'(rmc(s_1))$ if it satisfies the five conditions stated in the definition of AE' (see section 2.4.1, page 39). All conditions except condition (3.4e) follow directly from the definition of e' and the fact that $e \in AE(s_1)$. To prove that condition (3.4e) also holds, we use the fact that $F'_{\pi_1(e)}$ is defined such that the number of tokens produced by transition $\pi_1(e)$ in the uncoloured net (N') matches the number of tokens produced by $\pi_1(e)$ in N (see assumption) and the delay interval of a produced token in N is a sub-interval of the corresponding delay interval in N'. This and $\pi_3(e) \lhd \mathcal{BS}(F_{\pi_1(e)}(\mathcal{SB}(untime(\pi_2(e)))))$ imply that $\pi_3(e') \lhd \mathcal{BS}(F'_{\pi_1(e')}(\mathcal{SB}(untime(\pi_2(e')))))$, i.e. condition (3.4e) holds. See Odijk [94] for a more detailed proof.

(ii) Since rmc does not affect the timestamps of the tokens: $et(e') = et(e)$ and $tt(rmc(s_1)) = tt(s_1)$. Therefore, $et(e') = tt(rmc(s_1))$.

(iii) Because $e' = \langle \pi_1(e), rmc(\pi_2(e)), rmc(\pi_3(e)) \rangle$:
$(rmc(s_1) \setminus \pi_2(e')) \cup scale(\pi_3(e'), tt(rmc(s_1)))$
$= rmc(s_1) \setminus rmc(\pi_2(e)) \cup scale(rmc(\pi_3(e)), tt(s_1))$
$= rmc(s_1 \setminus \pi_2(e)) \cup rmc(scale(\pi_3(e), tt(s_1)))$

$= rmc((s_1 \setminus \pi_2(e)) \cup scale(\pi_3(e), tt(s_1)))$
$= rmc(s_2)$.
This completes our proof of this theorem.
□

A similar property also holds for the corresponding processes (Π and Π') generated by the two transition systems:

Lemma 18
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN and let $N' = (P', V', T', I', O', F', TS')$ be the corresponding uncoloured ITCPN. If $X = \langle S, R\rangle$ and $Y = \langle S', R'\rangle$ are the corresponding transition systems, then for any $s \in S$ and $\sigma \in \Pi(s)$, the 'uncoloured' path $\sigma' = \lambda_{i\in dom(\sigma)}\ rmc(\sigma_i)$ is a path in Y, i.e. $\sigma' \in \Pi'(rmc(s))$.

Proof.
Suppose that $\sigma \in \Pi(s)$ and $\sigma' = \lambda_{i\in dom(\sigma)}\ rmc(\sigma_i)$, then we have to prove that $\sigma' \in \Pi'(rmc(s))$. Hence, we prove that (see section 2.3):

(i) $0 \in dom(\sigma')$

(ii) $\sigma'_0 = rmc(s)$

(iii) $\forall_{i\in dom(\sigma')\setminus\{0\}}\ (i-1) \in dom(\sigma') \ \wedge\ \sigma'_{i-1} R' \sigma'_i$

(iv) $\forall_{i\in dom(\sigma')}\ (\forall_{j\in dom(\sigma')}\ j \leq i) \ \Rightarrow\ \sigma'_i \in S^{T'}$

(i) and (ii) follow directly from the definition of σ'. For any $i \in dom(\sigma')$: $(i-1) \in dom(\sigma')$, because $dom(\sigma') = dom(\sigma)$. Moreover, theorem 10 and $\sigma'_{i-1} = rmc(\sigma_{i-1})$, $\sigma'_i = rmc(\sigma_i)$ and $\sigma_{i-1} R \sigma_i$ imply that $\sigma'_{i-1} R' \sigma'_i$, hence (iii) holds. If $\sigma_i \in S^T$, then $\sigma'_i \in S^{T'}$, i.e. $R(\sigma_i) = \emptyset \ \Rightarrow R'(\sigma'_i) = \emptyset$, because $AE(\sigma_i) = \emptyset$ implies $AE'(rmc(\sigma_i)) = \emptyset$.
Hence, (iv) holds.
□

Theorem 10 and lemma 18 indicate that there is an interesting relationship between a net N and the corresponding uncoloured net N'. Note, there are some similarities with the soundness properties described in section 3.3.2 (recall, a morphism is also a similarity relation, see section 2.3).

We exploit theorem 10 and lemma 18 to show that it is possible to use the uncoloured net N' to prove certain properties of N. The uncoloured net N' can also be used to obtain bounds for performance measures like $\mathcal{EAT}_n$, $\mathcal{LAT}_n$, $\mathcal{LOR}$ and $\mathcal{HOR}$.

Lemma 19
Let N be an ITCPN and N' be the corresponding uncoloured ITCPN. For any initial state $s \in S$, we have that if N' is K-bounded w.r.t. $rmc(s)$, then N is K-bounded w.r.t. s.

Proof.
Use theorem 10.
□

Lemma 20
Let N be an ITCPN and N' be the corresponding uncoloured ITCPN. The transition system describing the semantics of N is $\langle S, R\rangle$. For any $s \in S$, we have that if N' is dead w.r.t. $rmc(s)$, then N is dead w.r.t. s.

Proof.
Use theorem 10.
□

Similar statements hold for transient, livelock free or (weakly) progressive nets.

Lemma 21
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN, the semantics of which is described by a transition system $X = \langle S, R\rangle$ and let $N' = (P', V', T', I', O', F', TS')$ be the corresponding uncoloured ITCPN, the semantics of which is described by a transition system $Y = \langle S', R'\rangle$. If $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$ are defined for N and $\mathcal{EAT}'_n$ and $\mathcal{LAT}'_n$ are defined for N' (see section 2.6), then for $s \in S$, $p \in P$ and $n \in \mathbb{N}$:

$$\begin{aligned} \mathcal{EAT}'_n(rmc(s), p) &\leq \mathcal{EAT}_n(s, p) \\ \mathcal{LAT}'_n(rmc(s), p) &\geq \mathcal{LAT}_n(s, p) \end{aligned}$$

Proof.
For any $\sigma \in \Pi(s)$ and $i \in dom(\sigma)$: $\mathrm{bmin}_n(\sigma_i \upharpoonright p) = \mathrm{bmin}_n(rmc(\sigma_i) \upharpoonright p)$. Lemma 18 tells us that $\sigma \in \Pi(s)$ implies that $\sigma' = (\lambda_{i \in dom(\sigma)}\ rmc(\sigma_i)) \in \Pi'(rmc(s))$. Therefore, $\mathcal{EAT}'_n(rmc(s), p) \leq \mathcal{EAT}_n(s, p)$ and $\mathcal{LAT}'_n(rmc(s), p) \geq \mathcal{LAT}_n(s, p)$ (see definition of $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$).
□

Lemma 22
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN and let $N' = (P', V', T', I', O', F', TS')$ be the corresponding uncoloured ITCPN. If $\mathcal{LOR}$ and $\mathcal{HOR}$ are defined for N and $\mathcal{LOR}'$ and $\mathcal{HOR}'$ are defined for N' (see section 2.6), then for $s \in S$, $p \in P$ and $x \in TS \setminus \{\infty\}$:

$$\begin{aligned} \mathcal{LOR}'_n(rmc(s), p, x) &\leq \mathcal{LOR}(s, p, x) \\ \mathcal{HOR}'_n(rmc(s), p, x) &\geq \mathcal{HOR}(s, p, x) \end{aligned}$$

Proof.
Use lemma 18.
□

These lemmas indicate that the approach which 'removes' all colouring may be useful. An important advantage of this method is that it produces, in a straightforward manner, an ITCPN which is easier to analyse. A disadvantage is the rigour of this approach, i.e. in most cases essential information is lost, thus making analysis useless. Consider for example the ITCPN shown in figure 3.19, although the analysis of the corresponding uncoloured net produces safe bounds for the performance of the ITCPN, these bounds are not as 'tight' as possible. Another disadvantage of this approach is the fact that it is not possible to answer questions involving the value of tokens, for example questions like 'What is the maximum number of tokens in place p_2 having a value l ?'.
The approach assumes that the number of tokens produced by the firing of a transition does not depend upon the values of the consumed tokens. This is not a necessary restriction, we restricted ourselves to this class of nets for reasons of simplicity. In Odijk [94] a construction is given which translates any ITCPN into an uncoloured ITCPN. Moreover, this problem will be addressed in the following subsection.
Note that an uncoloured net also allows for more traditional kinds of analysis like the calculation of siphons, traps and place and transition invariants. The results calculated for the uncoloured net N' can be interpreted for N, e.g. if $X \subseteq P$ is a siphon (trap) in N', then X is also is a siphon (trap) in N. In this way we can use Petri net theory, based on untimed uncoloured Petri nets, for our high-level Petri net model. A drawback of this approach is that these traditional kinds of analysis disregard all timing information in an ITCPN.

3.5.2 Approach 2: refine the net

The rigour of the first approach poses a number of problems if the delays and/or the number of tokens produced by a transition depend strongly on the values of the consumed tokens. To deal with these problems, we present an approach which decomposes some of the places into sets of places. A place p is decomposed into a number of places, say q_1,q_2 .. q_n, such that the value set of p is partitioned into the value sets of the places q_1,q_2 .. q_n. This is called a *refinement*. To refine a place p, we have to modify the input transitions and duplicate the output transitions. We refine the ITCPN until the 'desired' level of detail is visible in the network structure. Then we 'remove' all colouring, thus yielding an uncoloured net that can be analysed by the MTSRT method or some method based on uncoloured Petri nets. If we refine a net properly, we often obtain better analytic results. Compared to the first approach the latter approach is less rigourous.

Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN and let $N' = (P', V', T', I', O', F', TS')$ be the *refinement* of N with respect to a place $p_R \in P$, a set of new places Q and a function $D \in V_{p_R} \rightarrow Q$, notation: $N' = rf(N, p_R, Q, D)$. First, we show how to

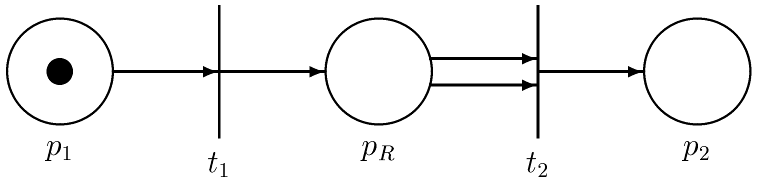


Figure 3.20: An ITCPN N

construct this N', then we will investigate the relation between these nets.

This refinement decomposes a place p_R into a number of new places. We assume that $Q \cap P = \emptyset$. We replace p_R by a set of places Q:

$$P' = (P \setminus \{p_R\}) \cup Q$$

The value set of an 'old' place (i.e. $p \in P \setminus \{p_R\}$) remains the same. Each new place has a value set that is a subset of V_{p_R}. Moreover, the value sets of the new places form a partitioning of the value set of p_R. Function D determines how V_{p_R} is partitioned:

$$\forall_{p \in P \setminus \{p_R\}} \quad V'_p = V_p$$
$$\forall_{p \in Q} \quad V'_p = \{v \in V_{p_R} \mid D(v) = p\}$$

Assume t is an output transition of p_R. It is not possible to map t to exactly one output transition $\hat{t}$ ($\#Q > 1$). If $\hat{t}$ is connected to each place in Q, this output transition remains disabled until all places of Q contain enough tokens. If $\hat{t}$ is not connected to a place p in Q and there are enough tokens in p ($I_t(p_R)$), then $\hat{t}$ may be disabled. Clearly this is not our intention. Therefore, each output transition of p_R has to be replaced by a number of 'new' transitions in such a way that at least one of these new transitions is enabled in N' if the corresponding transition in N is enabled. Consider for example the net N shown in figure 3.20. If p_R is decomposed into two places, q_1 and q_2, then the refined net contains three new transitions, say t_{21}, t_{22} and t_{23}, see figure 3.21. Transition t_2 in N is enabled if there are at least two tokens in p_R. Therefore, at least one of the transitions t_{21}, t_{22} t_{23} has to be enabled if there are at least two tokens in q_1 and q_2. Note that all other transitions in N (e.g. t_1) correspond to precisely one transition in the refined net.
A transition $t \in T$ with $I_t(p_R) = 1$ corresponds to exactly $\#Q$ transitions in N'. If $I_t(p_R) > 1$, then it is more difficult to calculate the corresponding number of transitions in the refined net. In this case, we have to count the number of ways in which it is possible to take precisely $I_t(p_R)$ tokens from $\#Q$ places, i.e. $\#\{m \in \mathbb{B}(Q) \mid \#m = I_t(p_R)\}$. Note that this number of ways equals:

$$\binom{\#Q + I_t(p_R) - 1}{\#Q - 1}$$

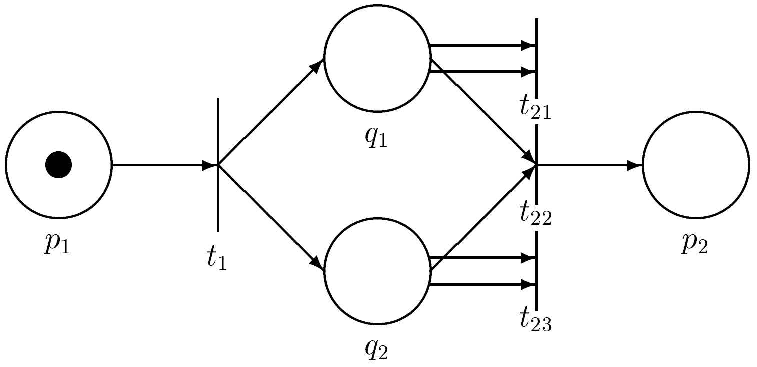


Figure 3.21: The refined net N'

In the example t_2 is replaced by $\begin{pmatrix} 2+2-1 \\ 2-1 \end{pmatrix} = \dfrac{3!}{1!(3-1)!} = 3$ transitions.
We name the new transitions as follows: $t \in p_R\bullet$ is replaced by a set of transitions identified by a pair $\langle t, m\rangle$, where m is a bag of places which specifies the number of tokens consumed from the 'new' places, i.e.

$$Y(t) = \{\langle t, m\rangle \mid m \in \mathbb{B}(Q) \ \wedge \ \#m = I_t(p_R)\}$$

$Y(t)$ is the set of transitions in N' which correspond to transition $t \in p_R\bullet$ in N.

$$T' = (T \setminus p_R\bullet) \ \cup \bigcup_{t \in p_R\bullet} Y(t)$$

Note that for all $t_1, t_2 \in T$: $Y(t_1) \cap T = \emptyset$ and $Y(t_1) \cap Y(t_2) = \emptyset$.
Given the new set of transitions T', the bag of input places of a transition in N' is defined rather straightforward:

$$\begin{aligned}
&\forall_{t \in T \setminus p_R\bullet} \quad I'_t = I_t \\
&\forall_{t \in p_R\bullet} \ \forall_{\hat{t} \in Y(t)} \quad I'_{\hat{t}} = (I_t \setminus \{\langle p_R, I_t(p_R)\rangle\}) \cup \pi_2(\hat{t})
\end{aligned}$$

A new transition $\langle t, m\rangle$ consumes $m(p)$ tokens from each 'new' place $p \in Q$ and $I_t(p)$ tokens from each 'old' place $p \in P \setminus \{p_R\}$.
The set of output places of each transition is defined as follows:

$$\begin{aligned}
&\forall_{t \in T \setminus (p_R\bullet \cup \bullet p_R)} \quad O'_t = O_t \\
&\forall_{t \in (T \setminus p_R\bullet) \cap \bullet p_R} \quad O'_t = (O_t \setminus \{p_R\}) \cup Q \\
&\forall_{t \in p_R\bullet \setminus \bullet p_R} \ \forall_{\hat{t} \in Y(t)} \quad O'_{\hat{t}} = O_t \\
&\forall_{t \in p_R\bullet \cap \bullet p_R} \ \forall_{\hat{t} \in Y(t)} \quad O'_{\hat{t}} = (O_t \setminus \{p_R\}) \cup Q
\end{aligned}$$

If a transition is not 'connected' to p_R, the set of output places remains the same. If a transition is an input transition of p_R, then the set of output places is modified

as follows: p_R is replaced by Q (if present). A 'new' transition 'inherits' the output places of the corresponding 'old' transition in N. If a 'new' transition is also an input transition of p_R, then p_R is replaced by Q.
There is no reason for adapting the time set, i.e.

$$TS' = TS$$

To define F', we introduce some conversion functions. The function $conv \in CT \to CT'$ converts an element of CT into an element of CT', i.e. for $\langle p, v\rangle \in CT$:

$$conv(\langle p, v\rangle) = \begin{cases} \langle p, v\rangle & \text{if } p \neq p_R \\ \langle D(v), v\rangle & \text{if } p = p_R \end{cases}$$

Note that $conv$ is a bijection. The functions $conv^{TS} \in (CT \times TS) \to (CT' \times TS')$ and $conv^{INT} \in (CT \times INT) \to (CT' \times INT')$ have similar definitions, i.e. for $\langle p, v\rangle \in CT$, $x \in TS$ and $w \in INT$:

$$\begin{aligned} &conv^{TS}(\langle\langle p, v\rangle, x\rangle) = \langle conv(\langle p, v\rangle), x\rangle \\ &conv^{INT}(\langle\langle p, v\rangle, w\rangle) = \langle conv(\langle p, v\rangle), w\rangle \end{aligned}$$

For convenience, we also define these functions for bags of tokens, e.g. if $b \in \mathbb{B}(CT)$, then $conv(b) = \lambda_{c \in CT'}\ b(conv^{-1}(c)) \in \mathbb{B}(CT')$. Now we are able to define F'. If $t \in T \setminus p_R\bullet$, then:

$$\begin{aligned} &dom(F'_t) = dom(F_t) \\ &\forall_{c \in dom(F'_t)}\ F'_t(c) = conv^{INT}(F_t(c)) \end{aligned}$$

Informally speaking, for the transitions not consuming tokens from Q, it suffices to convert the bag of produced tokens. If a transition $\hat{t}$ consumes tokens from a place in Q, i.e. there exists a $t \in T$ such that $\hat{t} \in Y(t)$, then the domain of $F'_{\hat{t}}$ has to be adapted.

$$\begin{aligned} &dom(F'_{\hat{t}}) = \{conv(c) \mid c \in dom(F_t)\ \wedge\ \lambda_{p \in P'}\ (\sum_{v \in V'_p} conv(c)(\langle p, v\rangle)\ ) = I'_{\hat{t}}\} \\ &\forall_{c \in dom(F'_{\hat{t}})}\ F'_{\hat{t}}(c) = conv^{INT}(F_{\pi_1(\hat{t})}(conv^{-1}(c)))) \end{aligned}$$

To clarify these rather formal notations, we refine the ITCPN shown in figure 3.19 with respect to the place p_1, the set $Q = \{q_0, q_1, q_2\}$ and the function $D \in \mathbb{N} \to Q$ such that for $n \in \mathbb{N}$:

$$D(n) = \begin{cases} q_0 & \text{if } n \bmod 3 = 0 \\ q_1 & \text{if } n \bmod 3 = 1 \\ q_2 & \text{if } n \bmod 3 = 2 \end{cases}$$

The refined net $N' = rf(N, p_1, Q, D)$ is shown in figure 3.22.

$P' = \{q_0, q_1, q_2, p_2, p_3, p_4\}$
$V'_{q_0} = \{n \in \mathbb{N} \mid n \bmod 3 = 0\}$,
$V'_{q_1} = \{n \in \mathbb{N} \mid n \bmod 3 = 1\}$,

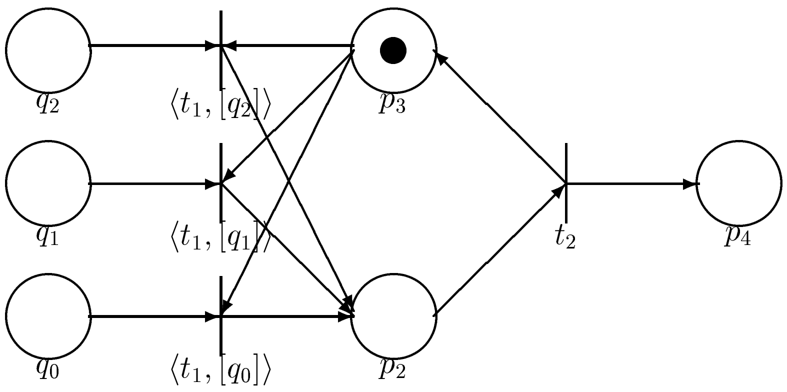

q_2
$\langle t_1,[q_2]\rangle$
p_3
q_1
$\langle t_1,[q_1]\rangle$
t_2
p_4
q_0
$\langle t_1,[q_0]\rangle$
p_2

Figure 3.22: The refined net $N' = rf(N, p_2, \{q_0, q_1, q_2\}, D)$

$V'_{q_2} = \{n \in \mathbb{N} \mid n \bmod 3 = 2\}$,
$V'_{p_2} = \mathbb{N}$, $V'_{p_3} = \{\emptyset\}$ and $V'_{p_4} = \mathbb{N}$
$T' = \{\langle t_1, [q_0]\rangle, \langle t_1, [q_1]\rangle, \langle t_1, [q_2]\rangle, t_2\}$
$I' = \{\langle\langle t_1, [q_0]\rangle, [q_0, p_3]\rangle, \langle\langle t_1, [q_1]\rangle, [q_1, p_3]\rangle, \langle\langle t_1, [q_2]\rangle, [q_2, p_3]\rangle, \langle t_2, [p_2]\rangle\}$
$O' = \{\langle\langle t_1, [q_0]\rangle, \{p_2\}\rangle, \langle\langle t_1, [q_1]\rangle, \{p_2\}\rangle, \langle\langle t_1, [q_2]\rangle, \{p_2\}\rangle, \langle t_2, \{p_3, p_4\}\rangle\}$
$TS' = TS$
$dom(F'_{\langle t_1,[q_0]\rangle}) = \{[\langle q_0, k\rangle, \langle p_3, \emptyset\rangle] \mid k \in \mathbb{N} \ \wedge \ k \bmod 3 = 0\}$
$dom(F'_{\langle t_1,[q_1]\rangle}) = \{[\langle q_1, k\rangle, \langle p_3, \emptyset\rangle] \mid k \in \mathbb{N} \ \wedge \ k \bmod 3 = 1\}$
$dom(F'_{\langle t_1,[q_2]\rangle}) = \{[\langle q_2, k\rangle, \langle p_3, \emptyset\rangle] \mid k \in \mathbb{N} \ \wedge \ k \bmod 3 = 2\}$
For $k \in \mathbb{N}$:
$F'_{\langle t_1,[q_0]\rangle}([\langle q_0, 3k\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, 3k\rangle, \langle 10, 15\rangle\rangle]$
$F'_{\langle t_1,[q_1]\rangle}([\langle q_1, 3k+1\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, 3k+1\rangle, \langle 11, 16\rangle\rangle]$
$F'_{\langle t_1,[q_2]\rangle}([\langle q_2, 3k+2\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, 3k+2\rangle, \langle 12, 17\rangle\rangle]$
$F'_{t_2} = F_{t_2}$

Informally speaking, a refinement reduces the size of a colour set (V_{p_R}) and moves information to the 'network level'. If we uncolour the refined net shown in figure 3.22, then we obtain the following ITCPN:

$P'' = \{q_0, q_1, q_2, p_2, p_3, p_4\}$
$V''_{q_0} = V''_{q_1} = V''_{q_2} = V''_{p_2} = V''_{p_3} = V''_{p_4} = \{\emptyset\}$
$T'' = \{\langle t_1, [q_0]\rangle, \langle t_1, [q_1]\rangle, \langle t_1, [q_2]\rangle, t_2\}$
$I'' = \{\langle\langle t_1, [q_0]\rangle, [q_0, p_3]\rangle, \langle\langle t_1, [q_1]\rangle, [q_1, p_3]\rangle, \langle\langle t_1, [q_2]\rangle, [q_2, p_3]\rangle, \langle t_2, [p_2]\rangle\}$
$O'' = \{\langle\langle t_1, [q_0]\rangle, \{p_2\}\rangle, \langle\langle t_1, [q_1]\rangle, \{p_2\}\rangle, \langle\langle t_1, [q_2]\rangle, \{p_2\}\rangle, \langle t_2, \{p_3, p_4\}\rangle\}$
$TS'' = TS$
$F''_{\langle t_1,[q_0]\rangle}([\langle q_0, \emptyset\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, \emptyset\rangle, \langle 10, 15\rangle\rangle]$
$F''_{\langle t_1,[q_1]\rangle}([\langle q_1, \emptyset\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, \emptyset\rangle, \langle 11, 16\rangle\rangle]$
$F''_{\langle t_1,[q_2]\rangle}([\langle q_2, \emptyset\rangle, \langle p_3, \emptyset\rangle]) = [\langle\langle p_2, \emptyset\rangle, \langle 12, 17\rangle\rangle]$
$F''_{t_2}([\langle p_2, \emptyset\rangle]) = [\langle\langle p_3, \emptyset\rangle, \langle 0, 0\rangle\rangle, \langle\langle p_4, \emptyset\rangle, \langle 0, 0\rangle\rangle]$

The reachability tree used by the MTSRT method to analyse the uncoloured refined net N'' contains only a few terminal states compared to the number of terminal states in the reduced reachability tree of N. To calculate $\mathcal{EAT}''_{50}(s, p_4) = 551$ and $\mathcal{LAT}''_{50}(s, p_4) = 801$, the MTSRT method has to evaluate a much smaller number of firing sequences. Note that these bounds are as 'tight' as possible. This example shows that an approach of a number of refinements followed by an 'uncolouring' can be very useful. Not every refinement is useful, perhaps even harmful in the sense that we may end up with less restrictive bounds. A successful refinement requires an intelligent selection of the places that have to be decomposed and a rational partitioning of the corresponding colour sets. Consider for example the net shown in figure 3.19, if we refine place p_2 into two places q_0 and q_1 for even and odd numbers respectively, then the refined net is not likely to give better analytic results.

We want to use the refined net N' to answer questions about the net N, therefore we have to establish a formal relationship between the corresponding transition systems. The transition systems are not equivalent, but there exists a very convenient morphism.

Theorem 11
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN, the semantics of which is described by a transition system $X = \langle S, R \rangle$ and let $N' = (P', V', T', I', O', F', TS')$ be the refined ITCPN with respect to a place p_R, a set Q and a function $D \in V_p \to Q$, i.e. $N' = rf(N, p_R, Q, D)$. The semantics of N' is described by a transition system $Y = \langle S', R' \rangle$. Now the function $dcp \in S \to S'$ is a morphism from X to Y, where dcp is defined as follows:

$$dom(dcp) = S$$
$$\forall_{s \in S}\ dcp(s) = \lambda_{i \in dom(s)}\ conv^{TS}(s(i))$$

Proof.
We confine ourselves to an outline of this proof. For any $s_1, s_2 \in S$ such that $s_1 R s_2$, we have to prove that $dcp(s_1) R' dcp(s_2)$. Because $s_1 R s_2$, there exists an event e such that:

(i) $e \in AE(s_1)$

(ii) $et(e) = tt(s_1)$

(iii) $s_2 = (s_1 \setminus \pi_2(e)) \cup scale(\pi_3(e), tt(s_1))$

Define $e' = \langle \hat{t}, dcp(\pi_2(e)), dcp(\pi_3(e)) \rangle \in E'$, where $\hat{t} \in Y(t)$ such that:
$\mathcal{SB}(untime(dcp(\pi_2(e)))) \in dom(F'_{\hat{t}})$.
Note that there is precisely one $\hat{t}$ satisfying these requirements.

Now it suffices to prove that:

(i) $e' \in AE'(dcp(s_1))$

(ii) $et(e') = tt(dcp(s_1))$

(iii) $dcp(s_2) = (dcp(s_1) \setminus \pi_2(e')) \cup scale(\pi_3(e'), tt(dcp(s_1)))$

Proving (i), (ii) and (iii) proceeds straightforwardly, but requires a lot of space. A formal proof of this theorem is given by Odijk in [94].
□

A property similar to the property of theorem 11, holds for the corresponding processes (Π and Π') generated by the two transition systems.

Lemma 23
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN, the semantics of which is described by a transition system $X = \langle S, R\rangle$ and let $N' = (P', V', T', I', O', F', TS')$ be the refined ITCPN with respect to a place p_R, a set Q and a function $D \in V_p \to Q$, i.e. $N' = rf(N, p_R, Q, D)$. The semantics of N' is described by a transition system $Y = \langle S', R'\rangle$.
Then for any $s \in S$ and $\sigma \in \Pi(s)$, the 'refined' path $\sigma' = \lambda_{i \in dom(\sigma)}\ dcp(\sigma_i)$ is a path in Y, i.e. $\sigma' \in \Pi'(dcp(s))$.

Proof.
Similar to the proof of lemma 18, use theorem 11.
□

Note that there is a lot of resemblance between these proofs and the proofs of theorem 10 and lemma 18. We can use theorem 11 and lemma 23 to obtain safe bounds for the performance measures defined in section 2.6. It is also possible to prove certain properties of the ITCPN via a refined ITCPN, for example boundedness and liveness properties.

Lemma 24
Let N be an ITCPN and N' be a refined ITCPN. The transition system describing the semantics of N is $\langle S, R\rangle$. For any $s \in S$: if N' is dead w.r.t. $dcp(s)$, then N is dead w.r.t. s.

Proof.
Use theorem 10.
□

Similar statements hold for transient, livelock free, bounded or (weakly) progressive nets. To interpret the analytic results obtained using the refined net, we have to extend the definition of $\mathcal{EAT}_n$, $\mathcal{LAT}_n$, $\mathcal{LOR}$ and $\mathcal{HOR}$ in a straightforward manner:

Definition 34 ($\mathcal{EAT}_n, \mathcal{LAT}_n$)
For an ITCPN, a set of states $A \subseteq S$, a set of places $Q \subseteq P$ and $n \in \mathbb{N} \setminus \{0\}$, we define:

$$\begin{aligned}
\mathcal{EAT}_n(A, Q) &= \min_{\sigma \in \Pi(A)} \min_{i \in dom(\sigma)} \text{bmin}_n(\sigma_i \upharpoonleft Q) \\
\mathcal{LAT}_n(A, Q) &= \max_{\sigma \in \Pi(A)} \min_{i \in dom(\sigma)} \text{bmin}_n(\sigma_i \upharpoonleft Q)
\end{aligned}$$

where for $s \in S$: $s \upharpoonleft Q = \lambda_{x \in TS}\ \#\{i \in dom(s) \mid place(s(i)) \in Q\ \wedge\ time(s(i)) = x\}$.

Lemma 25
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN, the semantics of which is described by a transition system $X = \langle S, R\rangle$ and let $N' = (P', V', T', I', O', F', TS')$ be the refined ITCPN with respect to a place p_R, a set Q and a function $D \in V_p \to Q$, i.e. $N' = rf(N, p_R, Q, D)$. The semantics of N' is described by a transition system $Y = \langle S', R'\rangle$. If $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$ are defined for N and $\mathcal{EAT}'_n$ and $\mathcal{LAT}'_n$ are defined for N', then for $s \in S$, $p \in P \setminus \{p_R\}$ and $n \in \mathbb{N}$:

$$\begin{aligned}
\mathcal{EAT}'_n(dcp(s), p) &\leq \mathcal{EAT}_n(s, p) \\
\mathcal{LAT}'_n(dcp(s), p) &\geq \mathcal{LAT}_n(s, p) \\
\mathcal{EAT}'_n(dcp(s), Q) &\leq \mathcal{EAT}_n(s, p_R) \\
\mathcal{LAT}'_n(dcp(s), Q) &\geq \mathcal{LAT}_n(s, p_R)
\end{aligned}$$

Proof.
Assume $p \in P \setminus \{p_R\}$.
For any $\sigma \in \Pi(s)$ and $i \in dom(\sigma)$: $\text{bmin}_n(\sigma_i \upharpoonleft p) = \text{bmin}_n(dcp(\sigma_i) \upharpoonleft p)$.
Lemma 23 tells us that $\sigma \in \Pi(s)$ implies that $\sigma' = (\lambda_{i \in dom(\sigma)}\ dcp(\sigma_i)) \in \Pi'(dcp(s))$.
Therefore, $\mathcal{EAT}'_n(dcp(s), p) \leq \mathcal{EAT}_n(s, p)$ and $\mathcal{LAT}'_n(dcp(s), p) \geq \mathcal{LAT}_n(s, p)$ (see definition of $\mathcal{EAT}_n$ and $\mathcal{LAT}_n$).
For any $\sigma \in \Pi(s)$ and $i \in dom(\sigma)$: $\text{bmin}_n(\sigma_i \upharpoonleft p_R) = \text{bmin}_n(dcp(\sigma_i) \upharpoonleft Q)$.
Therefore, we can also prove that $\mathcal{EAT}'_n(dcp(s), Q) \leq \mathcal{EAT}_n(s, p_R)$ and $\mathcal{LAT}'_n(dcp(s), Q) \geq \mathcal{LAT}_n(s, p_R)$.
$\Box$

Definition 35 ($\mathcal{LOR}, \mathcal{HOR}$)
If $s \in S$, $Q \subseteq P$ and $0 < t \in TS$, then we define:

$$\begin{aligned}
\mathcal{LOR}(s, Q, t) &= \min_{\sigma \in \Pi(s)} U(\sigma, Q, t) \\
\mathcal{HOR}(s, Q, t) &= \max_{\sigma \in \Pi(s)} U(\sigma, Q, t)
\end{aligned}$$

for the *lowest occupation rate* and *highest occupation rate* respectively, where U is extended in a straightforward manner (see section 2.6).

Lemma 26
Let $N = (P, V, T, I, O, F, TS)$ be an ITCPN, the semantics of which is described by a transition system $X = \langle S, R \rangle$ and let $N' = rf(N, p_R, Q, D)$ be the corresponding uncoloured ITCPN, the semantics of which is described by a transition system $Y = \langle S', R' \rangle$. If $\mathcal{LOR}$ and $\mathcal{HOR}$ are defined for N and $\mathcal{LOR}'$ and $\mathcal{HOR}'$ are defined for N', then for $s \in S$, $p \in P \setminus \{p_R\}$ and $x \in TS \setminus \{\infty\}$:

$$\begin{aligned}
\mathcal{LOR}'_n(dcp(s), p, x) &\leq \mathcal{LOR}(s, p, x)\\
\mathcal{HOR}'_n(dcp(s), p, x) &\geq \mathcal{HOR}(s, p, x)\\
\mathcal{LOR}'_n(dcp(s), Q, x) &\leq \mathcal{LOR}(s, p_R, x)\\
\mathcal{HOR}'_n(dcp(s), Q, x) &\geq \mathcal{HOR}(s, p_R, x)
\end{aligned}$$

Proof.
Use lemma 23.
□

These lemmas show that a refined net can be used to analyse the original net. The advantages of a refined net are straightforward: if we uncolour the refined net, we may improve the usefulness of the analytic results. A drawback is that this approach is often more time (and space) consuming than the first approach, but probably less so than analysing the original (coloured) net. Note that it is always possible to refine until the assumption of section 3.5.1 holds, i.e. the number of consumed tokens does not depend upon the values of the tokens consumed. In this way it is possible to uncolour any ITCPN.

We only considered refinements which decompose only one place. It is easy to extend this approach to allow a simultaneous decomposition of multiple places. Suppose we want to decompose two places with two refinements. The order in which these refinements take place does not matter (except for the naming of 'new' transitions). Moreover, a simultaneous refinement of these two places also yields an equivalent net.

An overview of the two approaches presented in this section is shown in figure 3.23. Suppose we have a question about a system modelled in terms of an ITCPN. We may try to analyse this net directly using the MTSRT method. This may lead to computational problems, since the reduced reachability graph is too large. To overcome this problem, we may decide to remove all colouring and apply analytic methods like the MTSRT method, the PNRT method, the ATCFN method or calculate the invariants of the net. Note that applying the PNRT method and the ATCFN method is not always possible (e.g. for a net with conflicts). An uncoloured net also allows for more traditional kinds of analysis like the calculation of siphons, traps and place and transition invariants.
In general, the results based on the analysis of the uncoloured net are not satisfactory, because they are not sufficiently detailed or the calculated bounds for the performance measures are rather trivial. To overcome these problems, we propose

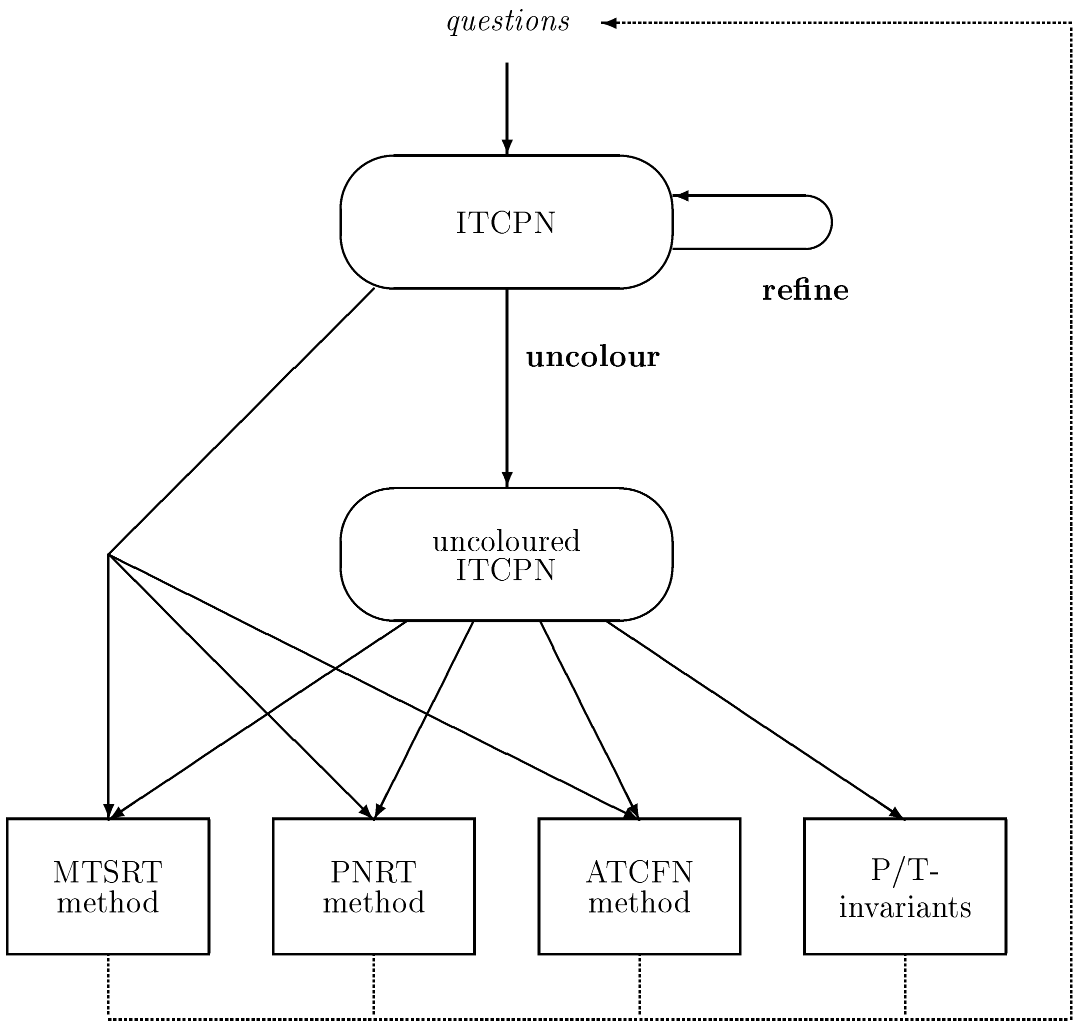


Figure 3.23: How to analyse an ITCPN

the following approach: first we decompose a number of places, i.e. refine the net, then we uncolour the refined net. Analysis of this uncoloured net will probably yield better results. If the results are still not satisfactory, then try some more refinements, .. etc.
Suppose we start with a net $N = (P, V, T, I, O, F, TS)$, this net is refined (in a number of steps) into a net $N' = (P', V', T', I', O', F', TS')$. Then we remove the remaining colouring and obtain the uncoloured net $N'' = (P'', V'', T'', I'', O'', F'', TS'')$. If we analyse N'', then the analytic results for this uncoloured refined net can be interpreted in terms of the original net N. For example, if N'' is bounded, then N is also bounded and, if N'' is dead, then N is also dead. Upper and lower bounds for various performance bounds of N'' are also valid for the original net N. For example, if $p \in P \cap P''$, then $\mathcal{EAT}''_n(s'', p) \leq \mathcal{EAT}_n(s, p)$ and $\mathcal{LAT}''_n(s'', p) \geq \mathcal{LAT}_n(s, p)$. The more we refine the net, the 'better' these bounds may become, but the larger the corresponding uncoloured ITCPN becomes, thus making analysis more time (and

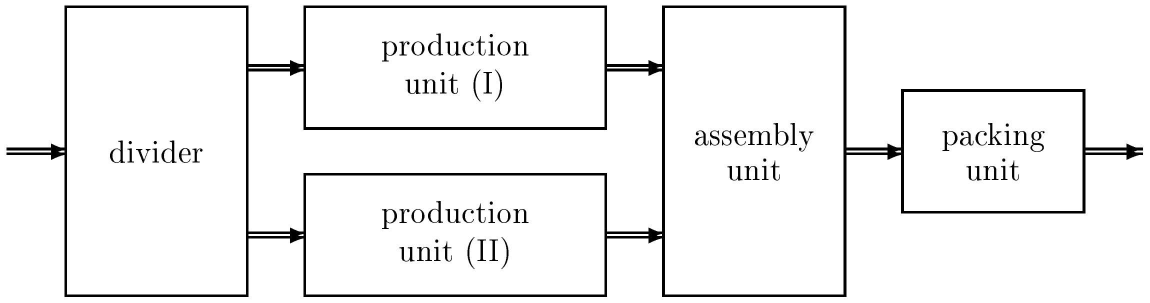


Figure 3.24: A manufacturing system

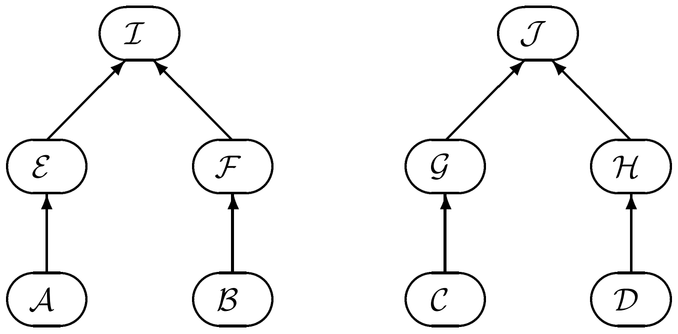


Figure 3.25: The bill of materials of end-products $\mathcal{I}$ and $\mathcal{J}$

space) consuming. In general, we have to balance between the quality of the results (e.g. how 'tight' the bounds are) and the effort it takes to analyse the net (e.g. computation time).

3.6 An example

We use an example to illustrate and demonstrate some of the concepts and techniques presented in this chapter. In this section we model and analyse a manufacturing system. This manufacturing system is divided into five units, see figure 3.24. The manufacturing system receives raw materials and transforms them into end-products. The raw materials are divided over two *production units*. Each production unit transforms raw materials into intermediate products. These intermediate products are assembled into end-products by the *assembly unit*. The *packing unit* prepares these products for shipment.
In this particular case, there are two kinds of end-products $\mathcal{I}$ and $\mathcal{J}$. To manufacture $\mathcal{I}$, we need two kinds of raw material: $\mathcal{A}$ and $\mathcal{B}$. $\mathcal{A}$ is transformed into $\mathcal{E}$, $\mathcal{B}$ is transformed into $\mathcal{F}$ and $\mathcal{E}$ and $\mathcal{F}$ are assembled into $\mathcal{I}$. $\mathcal{J}$ has a similar production process. The bill of materials of these two end-products is shown in figure 3.25.
We model this manufacturing process in terms of an ITCPN. This ITCPN has an

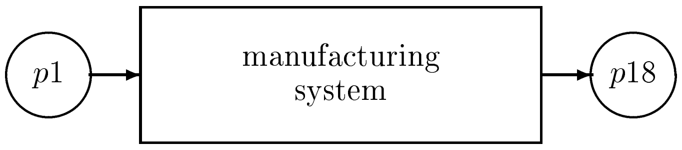


Figure 3.26: The interactions of the manufacturing system with the environment

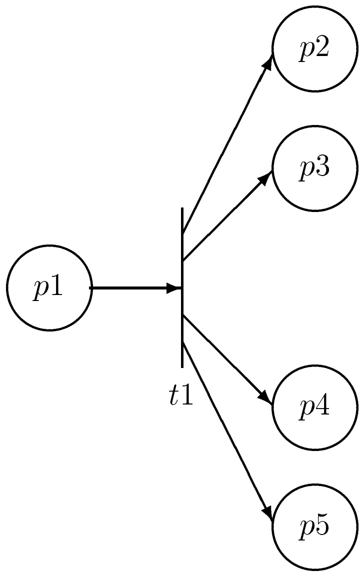


Figure 3.27: The divider

'input' place $p1$ to receive raw materials and an 'output' place $p18$ which contains end-products ready to be shipped. These two places are the only places having interactions with the environment of the manufacturing system, see figure 3.26. Tokens in these places represent products (or materials) and have a value which describes, the kind of product it represents, the identification of the product and some status information. Therefore, we define the value (colour) set of each place containing products (or material) as follows:

$$\begin{aligned} PT &= \{\text{`}\mathcal{A}\text{'}, \text{`}\mathcal{B}\text{'}, \text{`}\mathcal{C}\text{'}, \text{`}\mathcal{D}\text{'}, \text{`}\mathcal{E}\text{'}, ..\} \\ ID &= \mathbb{N} \\ STAT &= \mathbb{R} \\ V_{p1} &= V_{p18} = PT \times (ID \times STAT) \end{aligned}$$

The divider works as follows: it takes raw materials from place $p1$ and distributes them over the two production units. Moreover, the divider differentiates between the four kinds of raw material. Figure 3.27 shows the divider which is modelled by a transition $t1$ dividing the raw material over four places $p2$, $p3$, $p4$ and $p5$. The value sets of these places are equal to the value sets of the places $p1$ and $p18$, i.e. $V_{p2} = V_{p3} = V_{p4} = V_{p5} = PT \times (ID \times STAT)$. Transition $t1$ fires if there is some

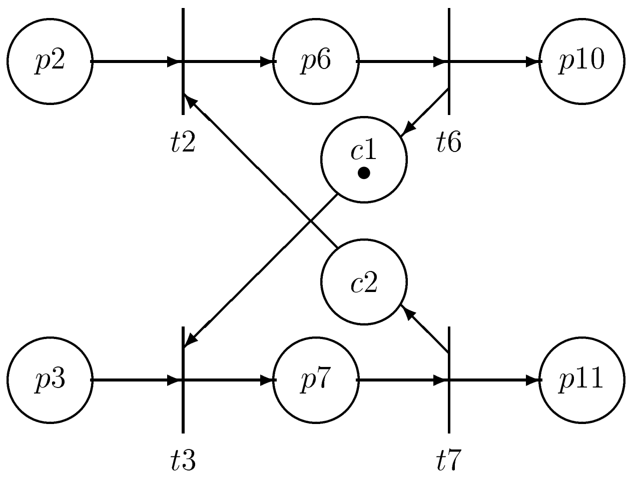


Figure 3.28: Production unit I

raw material available, material of kind $\mathcal{A}$ goes to place $p2$, material of kind $\mathcal{B}$ goes to place $p4$, material of kind $\mathcal{C}$ goes to place $p3$ and material of kind $\mathcal{D}$ goes to place $p5$. If $x \in PT \times (ID \times STAT)$, then:

$$F_{t1}([\langle p1, x\rangle]) = \begin{cases} [\langle\langle p2, x\rangle, \langle 0,0\rangle\rangle] & \text{if } \pi_1(x) = \text{`}\mathcal{A}\text{'} \\ [\langle\langle p3, x\rangle, \langle 0,0\rangle\rangle] & \text{if } \pi_1(x) = \text{`}\mathcal{C}\text{'} \\ [\langle\langle p4, x\rangle, \langle 0,0\rangle\rangle] & \text{if } \pi_1(x) = \text{`}\mathcal{B}\text{'} \\ [\langle\langle p5, x\rangle, \langle 0,0\rangle\rangle] & \text{if } \pi_1(x) = \text{`}\mathcal{D}\text{'} \end{cases}$$

Although the value sets of the places $p2$, $p3$, $p4$ and $p5$ are equal to V_{p1}, in this case they contain only one kind of products. Note that we assume that distributing these goods takes no time.

The first production unit transforms products of type $\mathcal{A}$ into $\mathcal{E}$ and products of type $\mathcal{C}$ into $\mathcal{G}$. These transformations are performed by one machine alternately working on products of type $\mathcal{A}$ and $\mathcal{C}$. This machine needs between 0.35 and 0.37 hours to transform $\mathcal{A}$ into $\mathcal{E}$ and between 0.78 and 0.81 hours to transform $\mathcal{C}$ into $\mathcal{G}$. Figure 3.28 shows this production unit in terms of an ITCPN. The machine has four states:

(i) busy, transforming $\mathcal{A}$ into $\mathcal{E}$

(ii) busy, transforming $\mathcal{C}$ into $\mathcal{G}$

(iii) free, waiting for product $\mathcal{A}$

(iv) free, waiting for product $\mathcal{C}$

Initially, the machine is in state (iv). In this example tokens in $c1$ and $c2$ are colourless ($V_{c1} = V_{c2} = \{\emptyset\}$) and the tokens in the other places represent products ($V_{p2} = V_{p3} = V_{p6} = V_{p7} = V_{p10} = V_{p11} = PT \times (ID \times STAT)$). The delay of a

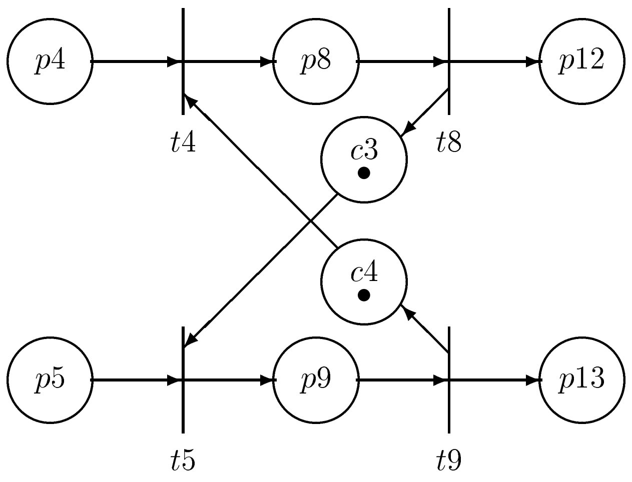


Figure 3.29: Production unit II

token produced by $t2$ is between 0.35 and 0.37, the delay of a token produced by $t3$ is between 0.78 and 0.81.

The second production unit has a similar structure, see figure 3.29. There are *two* identical machines, this is represented by the initial state, where there is one token in $c3$ and one token in $c4$. Both machines are capable of doing two kinds of transformations: $\mathcal{B}$ into $\mathcal{F}$ and $\mathcal{D}$ into $\mathcal{H}$. Transforming $\mathcal{B}$ into $\mathcal{F}$ takes between 1.58 and 1.61 hours. Transforming $\mathcal{D}$ into $\mathcal{H}$ takes between 0.18 and 0.20 hours. Initially, one of the machines is ready to transform $\mathcal{B}$ into $\mathcal{F}$, the other one is ready to transform $\mathcal{D}$ into $\mathcal{H}$.

There is one assembly unit. This unit is capable of assembling $\mathcal{E}$ and $\mathcal{F}$ into $\mathcal{I}$ and $\mathcal{G}$ and $\mathcal{H}$ into $\mathcal{J}$. Products are assembled in order of their arrival, i.e. the assembly unit uses a 'First Come First Served' discipline. The assembly unit consists of two dedicated assembly lines, one for end-product $\mathcal{I}$ and one for end-product $\mathcal{J}$. Figure 3.30 shows these two assembly lines. The assembly lines share a number of operators. Free operators are represented by tokens in the place o. Initially, there are five operators in the place o. To assemble $\mathcal{E}$ and $\mathcal{F}$ into $\mathcal{I}$ two operators are needed, this takes between 0.5 and 0.6 hours. The transition $t10$ consumes two tokens from place o and produces one token for place $p14$. Transition $t12$ produces two tokens for place $p14$ and one token for place $p16$. To assemble $\mathcal{G}$ and $\mathcal{H}$ into $\mathcal{J}$ three operators are needed, this takes between 1.3 and 1.4 hours. Transitions $t11$ and $t13$ represent the beginning and ending of this operation. Note that place $p16$ contains two kinds of products: $\mathcal{I}$ and $\mathcal{J}$.

The packing unit is used to prepare end-products $\mathcal{I}$ and $\mathcal{J}$ for shipment. To prepare these products, they are packed in wooden crates. Moreover, end-products $\mathcal{J}$ have to be tuned. The time needed to prepare a product for shipment depends on the

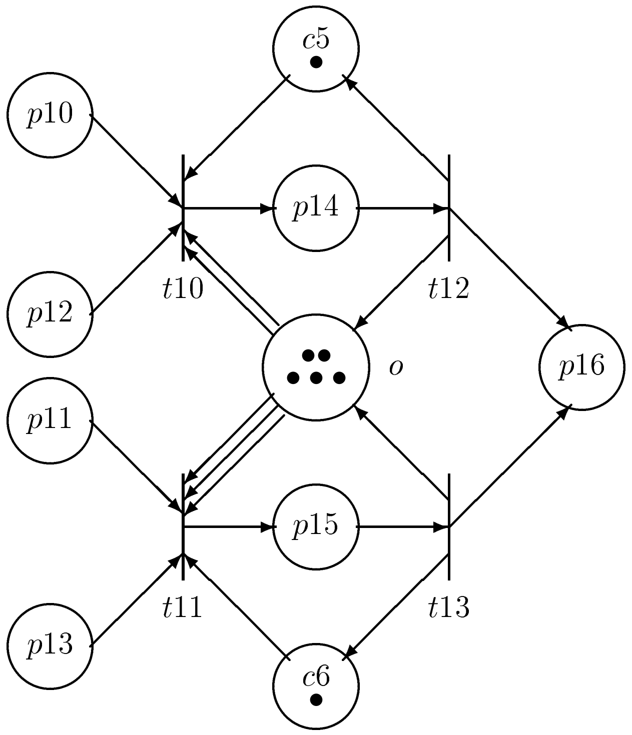


Figure 3.30: The assembly unit

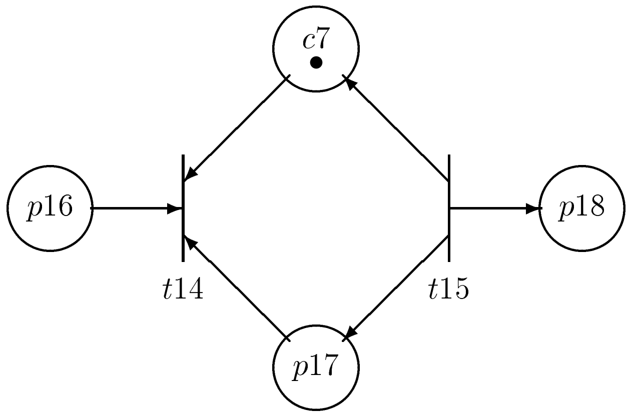


Figure 3.31: The packing unit

type of product: packing $\mathcal{I}$ takes between 0.2 and 0.3 hours, packing and tuning product $\mathcal{J}$ takes between 2.3 and 2.5 hours. The packing unit handles the products one by one. Figure 3.31 shows the corresponding net.

If we connect these units to each other, we get the ITCPN depicted in figure 3.32. Given this figure and the informal description already given, the formal definition $N = (P, V, T, I, O, F, TS)$ is rather straightforward.
The net N contains a conflict (see place o), therefore it is not possible to use the ATCFN or PNRT method. We can analyse this net directly using the MTSRT

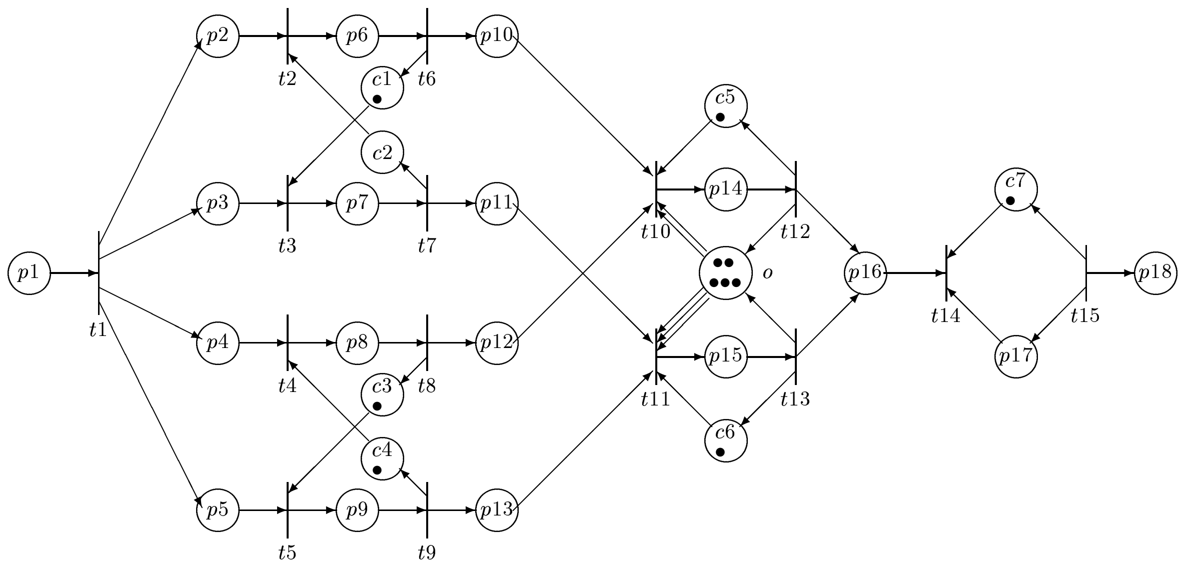


Figure 3.32: The ITCPN

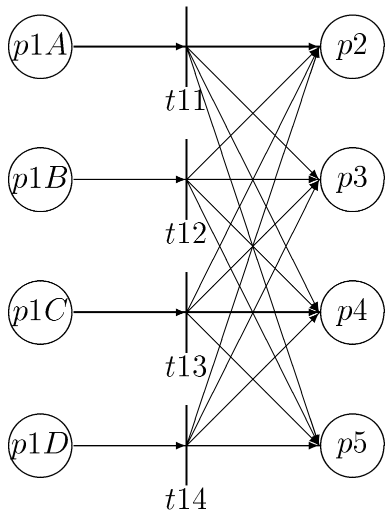


Figure 3.33: The divider in the refined net $rf(N, p1, Q, D)$

method. This is rather time consuming, since the appropriate software is lacking. Therefore, we uncolour the net. Uncolouring an ITCPN is always possible, see Odijk [94]. However, to uncolour the net as defined in section 3.5.1, we have to refine place $p1$, because the number of tokens produced by $t1$ for a specific output place depends on the value of the consumed token. Place $p1$ is decomposed into four places: $p1A$, $p1B$, $p1C$ and $p1D$. Note that we assume that there are only four kinds of raw material. Place $p1A$ contains tokens which represent raw material of type $\mathcal{A}$, place

n	$\mathcal{EAT}_n(s',p18)$	$\mathcal{LAT}_n(s',p18)$	$\mathcal{LAT}_n(s',p18) - \mathcal{EAT}_n(s',p18)$
1	1.48	3.91	2.43
2	2.61	6.41	3.80
3	3.08	8.91	5.83
4	3.74	11.41	7.67
5	4.38	13.91	9.53
6	4.87	16.41	11.54
7	5.68	18.91	13.23
8	6.00	21.41	15.41
9	6.98	23.91	16.93
10	7.18	26.41	19.23
11	8.26	28.91	20.65
12	8.46	31.41	22.95
13	9.39	33.91	24.52
14	9.59	36.41	26.82
15	10.88	38.91	28.03
16	12.18	41.41	29.23

Table 3.2: Some results produced by the MTSRT method

$p1B$ contains tokens which represent raw material of type $\mathcal{B}$, etc. In other words: we refine N with respect to the place $p1$, the set $Q = \{p1A, p1B, p1C, p1D\}$ and a function $D \in V_{p1} \rightarrow Q$ such that for $x \in V_{p1}$:

$$D(x) = \begin{cases} p1A & \text{if } \pi_1(x) = \text{`}\mathcal{A}\text{'} \\ p1B & \text{if } \pi_1(x) = \text{`}\mathcal{B}\text{'} \\ p1C & \text{if } \pi_1(x) = \text{`}\mathcal{C}\text{'} \\ p1D & \text{if } \pi_1(x) = \text{`}\mathcal{D}\text{'} \end{cases}$$

Figure 3.33 shows this refinement. Note that, although $t11$ is connected to $p3$, $p4$ and $p5$, it only produces tokens for place $p2$, i.e. we can omit the other arcs without affecting the behaviour of the net.
Now it is possible to uncolour the net, in the way it was described in the previous section. Assume that initially there are 32 pieces of raw material available (8 of each kind), i.e. in the initial state s there are 32 tokens in $p1$, eight with a value x such that $\pi_1(x) =$ '$\mathcal{A}$', .. etc. The corresponding uncoloured refined net has an initial state s' with eight tokens in place $p1A$, eight tokens in place $p1B$, eight tokens in place $p1C$ and eight tokens in place $p1D$. Using the MTSRT method we can calculate several performance measures, for example upper and lower bounds for the arrival time of tokens in place $p18$. Table 3.2 shows $\mathcal{EAT}_n(s',p18)$ and $\mathcal{LAT}_n(s',p18)$ for $1 \leq n \leq 16$, calculated using the MTSRT method. Based on these figures, we can guarantee that the 5^{th} end-product becomes available between

n	$\mathcal{EAT}_n(s'', p18)$	$\mathcal{LAT}_n(s'', p18)$	$\mathcal{LAT}_n(s'', p18)-$ $\mathcal{EAT}_n(s'', p18)$
1	1.48	1.71	0.23
2	2.61	2.89	0.28
3	5.18	5.51	0.33
4	5.38	8.01	2.63
5	5.58	10.51	4.93
6	6.00	13.01	7.01
7	8.08	15.51	7.43
8	8.28	18.01	9.73
9	8.48	20.51	12.03
10	9.39	23.01	13.62
11	10.98	23.31	12.33
12	13.28	23.61	10.33
13	15.58	24.20	8.62
14	17.87	24.51	6.64
15	20.17	24.81	4.64
16	22.47	25.11	2.64

Table 3.3: Some results produced by the MTSRT method

4.38 and 13.91. Note that the bounds calculated for this uncoloured net are quite 'wide'.

If we refine place $p16$ into $p16I$ and $p16J$, we obtain 'better' bounds, see table 3.3. This table shows some analytic results for the uncoloured refined net. This refinement decomposes place $p16$ into two places and transition $t14$ is split into two transitions, one for preparing end-products $\mathcal{I}$ and one for preparing end-products $\mathcal{J}$. The calculated bounds are more 'tight', because the preparation time in the packing unit is highly dependent of the kind of product ($\mathcal{I}$ or $\mathcal{J}$). This refinement is useful, but not totally satisfactory, because we are not able to distinguish between products $\mathcal{I}$ and $\mathcal{J}$ (see table 3.3).
Therefore, we also refine place $p17$ and place $p18$, place $p17$ is decomposed into $p17I$ and $p17J$, and place $p18$ is decomposed into $p18I$ and $p18J$. Figure 3.34 shows the refined net. Now we are able to calculate bounds for the completion time of the two kinds of end-products separately, see table 3.4. The results shown in this table are quite useful, e.g. based on these figures, we can *guarantee* that at time 20.00 there are 5, 6 or 7 products $\mathcal{J}$ available, etc. Note that we can use the techniques presented in this chapter to *prove* dynamic properties, e.g. the MTSRT method can be used to guarantee that certain deadlines are met.
We could have analysed the coloured net, shown in figure 3.32, directly (without uncolouring the net first). We did not do this, because we implemented the MTSRT method for uncoloured nets only, see chapter 4. To obtain useful results, the net

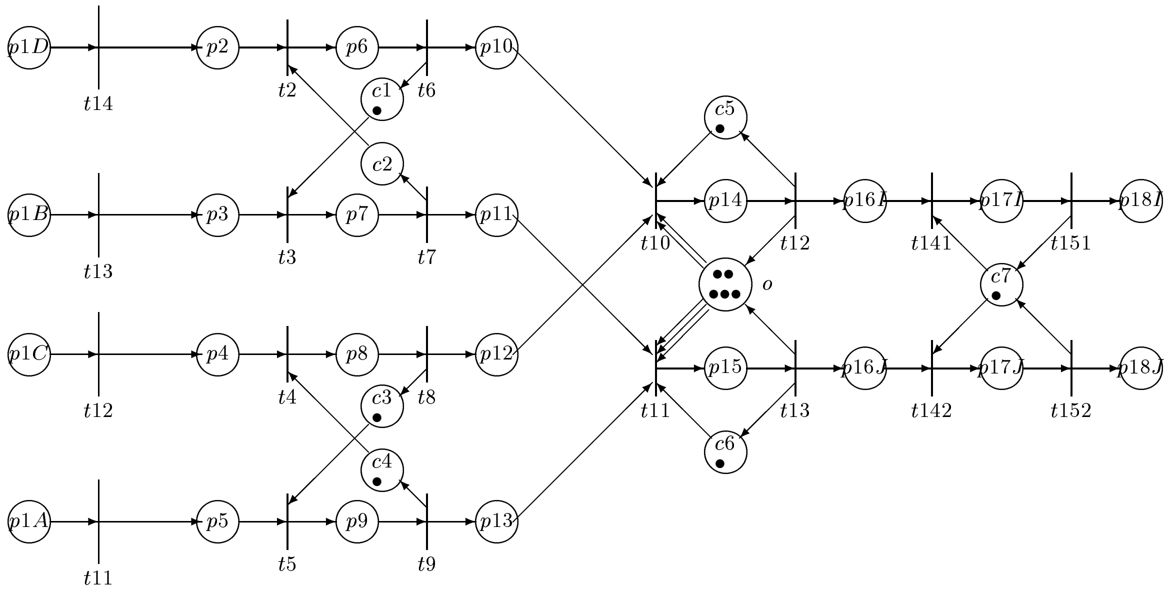


Figure 3.34: The refined ITCPN

should be refined properly before analysing the uncoloured net. The advantage of creating an uncoloured net is the fact that we can apply many analysis techniques based on uncoloured Petri nets without having an 'explosion' in the size of the net. Examples of Petri net based analysis techniques are place and transition invariants, traps, siphons and several reduction or decomposition techniques (see Murata [93]).

A place invariant $W \in P \rightarrow \mathbb{Z}$ is called a *minimal support invariant*, if and only if, W is non-negative (i.e. $\forall_{p \in P}\ W(p) \geq 0$) and there is no other non-negative place invariant $W' \in P \rightarrow \mathbb{Z}$, such that $\forall_{p \in P}\ W'(p) \leq W(p)$. The set of all minimal support place invariants can be used to generate the other invariants, i.e. any place invariant can be written as a linear combination of the minimal support place invariants. This property also holds for minimal support transition invariants (see Memmi and Roucairol [88] or Martinez and Silva [84]).
If we calculate the minimal support place invariants of the uncoloured ITCPN shown in figure 3.34, then we obtain the following results:

$p14 + c5 = 1$
$p15 + c6 = 1$
$2p14 + 3p15 + o = 5$
$p17I + p17J + c7 = 1$
$p6 + p7 + c1 + c2 = 1$
$p1 + p2 + p6 + p10 + p15 + p16J + p17J + p18J = 8$
$p1 + p3 + p7 + p11 + p14 + p16I + p17I + p18I = 8$
$p1 + p3 + p6 + p7 + p10 + p15 + p16J + p17J + p18J + c2 = 8$
$p1 + p2 + p6 + p7 + p11 + p14 + p16I + p17I + p18I + c1 = 9$

n	$\mathcal{EAT}_n(s''', p18I)$	$\mathcal{LAT}_n(s''', p18I)$	$\mathcal{LAT}_n(s''', p18I) - \mathcal{EAT}_n(s''', p18I)$
1	1.48	1.71	0.23
2	2.61	2.89	0.28
3	5.38	23.31	17.73
4	5.58	23.61	18.03
5	6.00	23.91	17.91
6	8.28	24.51	16.23
7	8.48	24.81	16.33
8	9.39	25.11	15.72

n	$\mathcal{EAT}_n(s''', p18J)$	$\mathcal{LAT}_n(s''', p18J)$	$\mathcal{LAT}_n(s''', p18J) - \mathcal{EAT}_n(s''', p18J)$
1	5.18	5.51	0.33
2	7.48	9.21	1.73
3	9.78	12.61	2.83
4	12.08	15.11	3.03
5	14.38	17.61	3.23
6	16.68	20.11	3.43
7	18.98	22.61	3.63
8	21.28	25.11	3.83

Table 3.4: Upper and lower bounds for the completion time of products $\mathcal{I}$ and $\mathcal{J}$

$p8 + p9 + c3 + c4 = 2$
$p1 + p4 + p8 + p12 + p15 + p16J + p17J + p18J = 8$
$p1 + p5 + p9 + p13 + p14 + p16I + p17I + p18I = 8$
$p1 + p5 + p8 + p9 + p12 + p15 + p16J + p17J + p18J + c4 = 9$
$p1 + p4 + p8 + p9 + p13 + p14 + p16I + p17I + p18I + c3 = 9$

The third place invariant ($2p14+3p15+o = 5$) indicates that the number of operators remains constant. The other invariants show that machines and products cannot get 'lost'. There are no transition invariants.

We have modelled and analysed some other examples using the approach presented in this chapter. A more detailed description of the application of this approach to production logistics and some examples are given by Odijk in [94]. Other examples can be found in Van der Aalst [2] and Van den Heuvel [61].

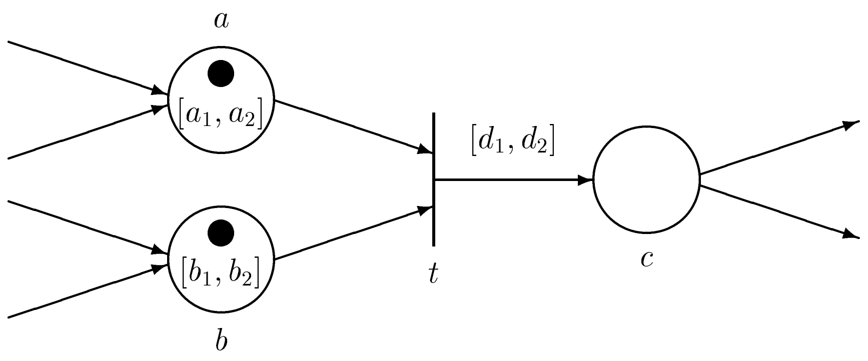


Figure 3.35: A part of some ITCPN

3.7 Conclusion

In this chapter we have introduced three new methods of analysis based on the interval timed coloured Petri net model developed in the previous chapter. These analysis techniques have been proved useful in the context of the questions raised in section 2.6.
The ATCFN method distinguishes itself by its simplicity. Although the ATCFN method has a number of serious drawbacks and limitations, it can be used in the discipline called project engineering (see section 3.2.1).
The ATCFN method cannot be used to analyse complex systems with repetitive events, such as logistic systems and production systems. The MTSRT method is a much more powerful method, since it can be applied to arbitrary nets and answers a large variety of questions. The method generates a reduced reachability graph. Even for small timed coloured Petri nets this reachability graph tends to become too large if there are a lot of conflicts or the colour sets are very large.
For a subclass of nets (marked graphs satisfying some additional constraints), we can use the PNRT method to analyse the net more efficiently.
To deal with large colour sets, we propose the two approaches described in section 3.5.

Because the duration of each delay is specified by an interval, the analysis techniques presented in this chapter produce upper and lower bounds for performance measures like throughput time and occupation rate. Consider for example the refined net shown in figure 3.34. Analysis shows that the completion time of the 7^{th} product of type $\mathcal{J}$ is between 18.98 and 22.61 (see table 3.4). If the specified delay intervals are safe, then the calculated bounds are guaranteed to be safe.
However, there is a problem if we have to estimate some of the delay intervals. Another possibility is that we deliberately shorten the length of an interval to test the sensitivity of the calculated bounds. In both cases we are interested in the risk of calculating inaccurate bounds.

For this purpose, we consider a typical situation, shown in figure 3.35. This figure shows a part of some ITCPN. We will use this subnet to reason about the sensitivity of the results produced by the MTSRT method.
A possible state in the reduced reachability graph generated by the MTSRT method is the following one: there is one token in place a with time interval $[a_1, a_2]$ and there is one token in place b with time interval $[b_1, b_2]$ (see figure 3.35). Assume that we know that the actual delay of the token produced by transition t is in $[d_1, d_2]$ with probability $1 - p_d$. Also assume that the time intervals of the tokens in the places a and b are the result of 'unsafe' delay intervals, i.e. the actual timestamp of the token in a is in $[a_1, a_2]$ with probability $1 - p_a$ and the actual timestamp of the token in b is in $[b_1, b_2]$ with probability $1 - p_b$. Furthermore, assume that t is the only enabled transition. In this case, transition t will fire, the resulting state in the reduced reachability graph has a token in c with a time interval $[c_1, c_2]$, where $c_1 = (a_1 \max b_1) + d_1$ and $c_2 = (a_2 \max b_2) + d_2$. Now we are interested in the probability that the actual timestamp of this token is in this time interval calculated by the MTSRT method.
More formally, if X_a is a random variable representing the actual timestamp of the token in a, X_b is a random variable representing the actual timestamp of the token in b and X_d is a random variable representing the actual delay of the token produced by t, then we are interested in the random variable $X_c = (X_a \max X_b) + X_d$, i.e. the actual timestamp of the token in c. We know that $p_a = \mathbb{P}[X_a \notin [a_1, a_2]]$, $p_b = \mathbb{P}[X_b \notin [b_1, b_2]]$, $p_d = \mathbb{P}[X_d \notin [d_1, d_2]]$ and are interested in $p_c = \mathbb{P}[X_c \notin [c_1, c_2]]$.

$$\begin{array}{ll} & \mathbb{P}[X_c \notin [c_1, c_2]] \\ = & \mathbb{P}[X_c \notin [(a_1 \max b_1) + d_1, (a_2 \max b_2) + d_2]] \\ = & 1 - \mathbb{P}[X_c \in [(a_1 \max b_1) + d_1, (a_2 \max b_2) + d_2]] \\ \leq & 1 - \mathbb{P}[X_a \in [a_1, a_2] \ \wedge \ X_b \in [b_1, b_2] \ \wedge \ X_d \in [d_1, d_2]] \\ = & 1 \ - \ (1 - p_a)(1 - p_b)(1 - p_d) \\ \leq & p_a + p_b + p_d \end{array}$$

If we add the extra assumption that the lower bounds of the intervals are safe, i.e. $\mathbb{P}[X_a \geq a_1] = \mathbb{P}[X_b \geq b_1] = \mathbb{P}[X_d \geq d_1] = 1$, then we deduce:

$$\begin{array}{ll} & \mathbb{P}[X_c \notin [c_1, c_2]] \\ = & \mathbb{P}[X_c \notin [(a_1 \max b_1) + d_1, (a_2 \max b_2) + d_2]] \\ = & \mathbb{P}[X_c > (a_2 \max b_2) + d_2] \\ \geq & \mathbb{P}[X_a > a_2 \ \wedge \ X_b > b_2 \ \wedge \ X_d > d_2] \\ = & p_a p_b p_d \end{array}$$

Note that we also assumed that $\mathbb{P}[X_a = a_2] = \mathbb{P}[X_b = b_2] = \mathbb{P}[X_d = d_2] = 0$. A similar deduction holds for safe upper bounds instead of safe lower bounds. In both cases we conclude:

$$p_a p_b p_d \leq p_c = \mathbb{P}[X_c \notin [c_1, c_2]] = p_a + p_b + p_d$$

Suppose $p_a = p_b = p_d = 0.1$, then $p_a p_b p_d = 0.001$ and $p_a + p_b + p_d = 0.3$, i.e. $0.001 \leq p_c \leq 0.3$. Obviously these figures do not tell us much. If p_c is near 0.001 the effect of 'unsafe' intervals will fade away. On the other hand, if p_c is near 0.3 the effect of several 'unsafe' delay intervals is amplified, i.e. the error probabilities add up.

To obtain more information about p_c, we need to know more about the distribution of the random variables X_a, X_b and X_d.

These results show that we have to be very careful when deciding on the delay intervals. This may seem disappointing, but it also indicates that the calculated bounds are far from trivial, because assuming an 'unsafe' delay interval makes the calculated bounds more 'tight', but also unreliable.

We also investigated the probability distribution of performance measures like the arrival time of the n^{th} token in a place p under the assumption that all delays are sampled from some probability distribution (e.g. a uniform distribution). Experimentation shows that the probability distribution of such a performance measure depends on the specific situation and it is impossible to make general statements. Consider for example the location of the probability mass. Sometimes it is mainly in the middle of the calculated interval, at other times an important part is near one of the borders of the interval $[\mathcal{EAT}_n(s,p), \mathcal{LAT}_n(s,p)]$.

There are several factors which prevent us from making a reasonable prediction of the shape of the probability density function of such a performance measure.

In the ITCPN model, conflicts between transitions are resolved non-deterministically. If we assume a probability distribution associated with the choice of the transition to be fired among several enabled transitions, then a conflict between transitions may result in a probability density function which contains multiple local maxima.

If at some moment one of the input places contains an abundant number of tokens having overlapping time intervals, then the probability mass of the distribution of the timestamp of a produced token is shifted towards the lower bound of the calculated interval. In other words, conflicts between tokens may shift the probability mass towards the lower bound of the interval.

Other phenomena effecting the shape of the probability density function of a performance measure like the arrival time of the n^{th} token in a place, are dependencies between tokens and feedbacks.

Again these results may seem disappointing, but they indicate three features. First of all, the calculated bounds are non-trivial because the probability mass may be near one of the bounds. Secondly, they show us that it is not possible to predict the distribution of performance measures without assuming very specific delay distributions (e.g. negative exponential distributed delays). Thirdly, the use of interval timing allows for the answering of a meaningful but limited set of questions. If we are really interested in characteristics like the means and variances of certain performance measures, then we should use other techniques like simulation or stochastic analysis (e.g. Markovian analysis based on a stochastic Petri net model, see section 1.4).

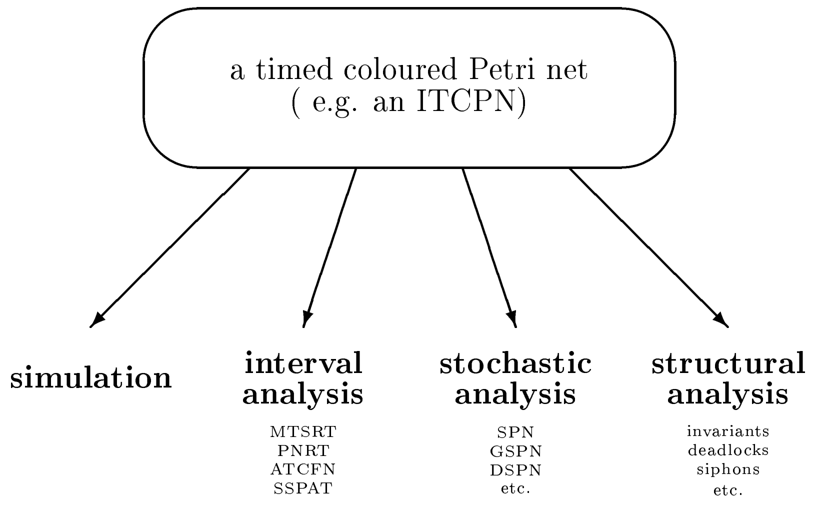


Figure 3.36: A survey of potential methods for the analysis of ITCPNs

Based on these observations we propose a situation where interval analysis is used in combination with simulation and other Petri net based analysis techniques, see figure 3.36. In this figure we distinguish four kinds of analysis: simulation, 'interval analysis', 'stochastic analysis' (e.g. stochastic Petri nets) and 'structural analysis'. All these analysis techniques can be used to analyse a timed coloured Petri net, for example an ITCPN. Note that if we use the ITCPN model, then we have to supply extra information to simulate the net (this also holds for stochastic analysis). On the other hand, most kinds of structural analysis do not require timing information.
Simulation can be used in the modelling phase and the (performance) analysis phase. In the modelling phase simulation can reveal errors, i.e. it can be used to 'debug' the model. In the analysis phase, simulation can be used to investigate the performance of the system. Stochastic Petri nets are also used to investigate the performance of the system. Compared to simulation, Markovian analysis of these nets is faster but it requires more proficiency and we have to assume that the delays are sampled from a rather specific probability distribution. Interval analysis can be used in the early investigation of the performance of the system, because it requires less information than Markovian analysis and simulation. It can also be used to prove the (temporal) correctness of the system. Structural analysis is mainly used to validate the logical correctness of the system.

Note that the ITCPN model can be used as a 'blueprint' of the system, which allows for various kinds of analysis. This is very convenient, since it prevents us from having to remodel the system every time we want to use another analysis technique. We are also interested in supporting other analysis techniques, e.g. queueing networks, linear programming, etc. (see chapter 5).
An ideal situation is the following one: there is one model that can be analysed by

several analysis techniques without having to change the model. In order to use several kinds of analysis at the same time, it is necessary to develop software tools based on one central timed coloured Petri net model (e.g. the ITCPN model extended with stochasticity). In chapter 4 we describe the software we have developed to realize this goal.

Finally, we conclude with our plans for future research in this area. To handle very large systems, we have to add more reduction techniques to the MTSRT method. Several examples show that conflicts between transitions often cause computational problems. We have a number of ideas to prevent this from happening (e.g. to aggregate states having the same marking and consecutive intervals). We are also interested in extending existing analysis methods for our ITCPN model. For example, we are convinced that it is possible to extend the analysis method proposed by Berthomieu et al. in [17] and [16].
Another item for further research is the use of *perturbation analysis* for the analysis of timed coloured Petri nets (see Ermoliev, Uryas'ev and Wessels [40]). One of the disadvantages of simulation and most of the other analysis techniques described in this chapter is that they only evaluate one scenario, without giving much help in finding better scenarios. Perturbation analysis is a method which estimates the gradient of some performance measure with respect to a parameter θ, based on a simulation run for only one value of θ only (see Suri [118]). Although perturbation analysis is still in its infancy, it might provide techniques which help us to find better scenarios, i.e. a better design (see Ermoliev, Uryas'ev and Wessels [40]).

3.8 Appendix

In this appendix, we present lemma 27 which is used to prove theorem 6 (see section 3.3.2).

In section 2.4.1 we have defined the semantics of an ITCPN. These semantics are such that tokens are consumed in order of their timestamps (see requirement (2.4c) on page 39).
However, in the modified transition system, (only) tokens having the same value are consumed in non-descending order (see requirement (3.4c) on page 84).
In order to prove theorem 6, we investigate this difference.

Assume $s_1 \in S$ and $\overline{s}_1 \in \overline{S}$ such that $s_1 \lhd \overline{s}_1$, and e is an event transforming s_1 into s_2 (i.e. $s_2 \in R(s_1)$).
We have to show that for any e transforming s_1 into s_2 there exists a 'corresponding' event $\overline{e}$, such that tokens having the same value are consumed in non-descending order.
Since $s_1 \lhd \overline{s}_1$, there exists a specialization function f, i.e. there exists a bijective function $f \in \mathit{dom}(s_1) \rightarrow \mathit{dom}(\overline{s}_1)$ such that every token with label $i \in \mathit{dom}(s_1)$ corresponds to a token with label $f(i) \in \mathit{dom}(\overline{s}_1)$ that is in the same place, has the same value and has an interval containing the timestamp of i.
Define $\overline{e} = \langle \pi_1(e), \overline{s}_1 \restriction f(\mathit{dom}(\pi_2(e))), q\rangle \in \overline{E}$, as in the proof of theorem 6.
Now we have to prove that we can change f into an 'order-preserving' function g satisfying the same constraints, i.e for all $i \in \mathit{dom}(\pi_2(\overline{e}))$ and $j \in \mathit{dom}(\overline{s}_1) \setminus \mathit{dom}(\pi_2(\overline{e}))$, such that $\mathit{place}(\overline{s}_1(i)) = \mathit{place}(\overline{s}_1(j))$ and $\mathit{value}(\overline{s}_1(i)) = \mathit{value}(\overline{s}_1(j))$, we have that $\neg(\mathit{time}(\overline{s}_1(j)) <_i \mathit{time}(\overline{s}_1(i)))$

For simplicity, we consider only one place, say p. For tokens having a different value, requirement (3.4c) holds. Therefore, we concentrate on tokens having the same value, i.e. assume that all tokens in p have an identical value.
Let $q \in Id \not\rightarrow TS$ represent the contents of place p in state s_1. Let $\overline{q} \in Id \not\rightarrow INT$ represent the contents of place p in state $\overline{s}_1$.
Since $s_1 \lhd \overline{s}_1$, there exists a bijective function $f \in \mathit{dom}(q) \rightarrow \mathit{dom}(\overline{q})$ such that for all $i \in \mathit{dom}(q)$, we have that $q(i) \in \overline{q}(f(i))$.

Lemma 27 (Assignment Problem)
If $q \in Id \not\rightarrow TS$ and $\overline{q} \in Id \not\rightarrow INT$ such that there exists a function $f \in \mathit{dom}(q) \rightarrow \mathit{dom}(\overline{q})$ with:

(i) f is bijective

(ii) $\forall_{i \in \mathit{dom}(q)}\ q(i) \in \overline{q}(f(i))$

then there also exists a function $g \in \mathit{dom}(q) \rightarrow \mathit{dom}(\overline{q})$ with:

(iii) g is bijective

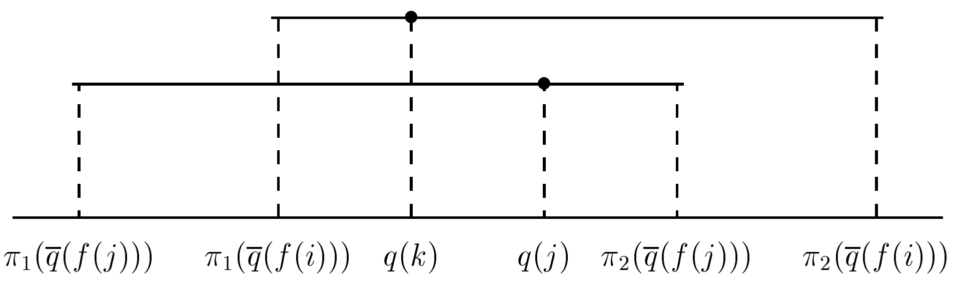

$\pi_1(\overline{q}(f(j)))$
$\pi_1(\overline{q}(f(i)))$
$q(k)$
$q(j)$
$\pi_2(\overline{q}(f(j)))$
$\pi_2(\overline{q}(f(i)))$

Figure 3.37: Interval $\overline{q}(f(i))$ and interval $\overline{q}(f(j))$

(iv) $\forall_{i \in dom(q)}\ q(i) \in \overline{q}(g(i))$

(v) $\forall_{i,j \in dom(q)}\ q(i) \leq q(j) \Rightarrow \neg(\overline{q}(g(j)) <_i \overline{q}(g(i)))$

Proof.
It is easy to find a function g that satisfies (iii) and (iv), because f is such a function. In this proof we will show that it is possible to 'transform' f until (v) holds (i.e. we give an algorithm to calculate g). First, we define a linear (total) ordering ($\leq_l$) on $dom(q)$ such that $i \leq_l j \Rightarrow q(i) \leq q(j)$. This is possible, because $q(i) \leq q(j)$ defines a pre-ordering (a pre-ordering (quasi-ordering) is reflexive and transitive).
Now we are able to define the *conflict set* of f:

$$C(f) = \{\langle i,j \rangle \in dom(q) \times dom(q) \mid i \leq_l j \ \wedge \ \overline{q}(f(i)) >_i \overline{q}(f(j))\}$$

Note that $C(f) = \emptyset$ implies that $\forall_{i,j \in dom(q)}\ q(i) \leq q(j) \Rightarrow \neg(\overline{q}(f(i)) >_i \overline{q}(f(j)))$.
Consider the following program to transform f (in pseudo code):

```
while  $C(f) \neq \emptyset$
begin
       $\langle i,j \rangle \in C(f)$
       { select an $i$ and $j$ in conflict }
       $f := (f \upharpoonright (dom(q) \setminus \{i,j\}) \cup \{\langle i, f(j) \rangle, \langle j, f(i) \rangle\}$
       { swap $i$ and $j$ }
end
```

Because, $C(f) = \emptyset$ implies (v), it is sufficient to prove that (iii) and (iv) are invariant and that the program terminates.

First, we prove that (iii) and (iv) are invariant. Initially, both invariants hold, because of the definition of f. Suppose (iii) and (iv) hold and $\langle i,j \rangle \in C(f)$ and
$\hat{f} := (f \upharpoonright (dom(q) \setminus \{i,j\}) \cup \{\langle i, f(j) \rangle, \langle j, f(i) \rangle\}$
Now we have to show that both invariants hold for $\hat{f}$.

If f bijective, then $\hat{f}$ also bijective ((iii) holds).

To prove (iv), we have to show that for any $k \in dom(q)$: $q(k) \in \overline{q}(\hat{f}(k))$.
(a) If $k \neq i$ and $k \neq j$, then $q(k) \in \overline{q}(f(k)) = \overline{q}(\hat{f}(k))$.
(b) If $k = i$, then $q(i) \in \overline{q}(f(i)) = \overline{q}(\hat{f}(j))$.
We also know that $q(i) \leq q(j)$ and $\overline{q}(f(i)) >_i \overline{q}(f(j))$, because $\langle i, j \rangle \in C(f)$.
The fact that $\overline{q}(f(i)) >_i \overline{q}(f(j))$ implies that $(\pi_1(\overline{q}(f(i))) \geq \pi_1(\overline{q}(f(j))))$ and $(\pi_2(\overline{q}(f(i))) \geq \pi_2(\overline{q}(f(j))))$. This situation is shown in the following figure 3.37.
$q(k) \geq \pi_1(\overline{q}(f(k))) \geq \pi_1(\overline{q}(f(j))) = \pi_1(\overline{q}(\hat{f}(k)))$
$q(k) \leq q(j) \leq \pi_2(\overline{q}(f(j))) = \pi_1(\overline{q}(\hat{f}(k)))$
So $q(k) \in \overline{q}(\hat{f}(k))$.
(c) A similar reasoning holds for $k = j$.

Finally, we have to prove that the program terminates. Observe that there are only a finite number of bijective functions from $dom(q)$ to $dom(\overline{q})$ $((\#dom(q))!)$.
Using the linear ordering $\leq_l$ it is possible to construct a lexicographic ordering ($\leq_f$) on the set of functions from $dom(q)$ to $dom(\overline{q})$: If $f, f' \in dom(q) \rightarrow dom(\overline{q})$, then:

$$\begin{aligned} f \leq_f f' \quad \equiv \quad & \exists_{k \in dom(q)} (\forall_{\substack{l \in dom(q) \\ l <_l k}} f(l) = f'(l)) \ \wedge \ \overline{q}(f(k)) <_i \overline{q}(f'(k))) \quad \vee \\ & \forall_{k \in dom(q)} \ f(k) = f'(k) \end{aligned}$$

This ordering is a partial ordering, because $\leq_i$ is a partial ordering. It is easy to verify that $\leq_f$ is reflexive and antisymmetric ($\leq_i$ is antisymmetric). The ordering is also transitive: $f \leq_f f'$ and $f' \leq_f f''$ implies that $f \leq_f f''$ ($\leq_i$ is transitive).

If $\hat{f}$ is the result of swapping i and j in f, then $\hat{f} <_f f$, because $\forall_{\substack{l \in dom(q) \\ l <_l i}} \hat{f}(l) = f(l)$ and $\overline{q}(\hat{f}(i)) <_i \overline{q}(f(i)))$.

The fact that f is 'descending' with respect to $\leq_f$ and that the number of possible functions is finite tells us that the algorithm will terminate. Therefore, there exists a function g that satisfies the conditions (iii),(iv) and (v).
□

Suppose, event e consumes the tokens with a label in $X \subseteq dom(q)$ from place p. Because of requirement (2.4c), we know that:

$$\forall_{i \in X} \ \forall_{j \in dom(q) \backslash X} \quad q(i) \leq q(j)$$

Using lemma 27, we deduce that there exists a g such that (iii), (iv) and (v) hold. This implies that:

$$\forall_{i \in g(X)} \ \forall_{j \in dom(\overline{q}) \backslash g(X)} \quad \neg(\overline{q}(j) <_i \overline{q}(i))$$

Consequently, there exists an $\overline{e}$ such that requirement (3.4c) holds.

The modified transition system consumes tokens having a different value in a non-deterministic manner (see requirement (3.4c) on page 84). Consider for example a

place p containing two tokens having different values. One of these tokens has a time interval v and the other one has a time interval w. If t is an output transition of p ($I_t(p) = 1$) and t is enabled, then there are at least two allowed events (one for each token), no matter how the time intervals v and w are related. If, for example, $v = \langle 1, 3\rangle$ and $w = \langle 4, 6\rangle$ (i.e. all timestamps in v are smaller than any timestamp in w), then the modified transition system considers the event consuming the token with timestamp w an allowed event.
To avoid this, we can add an extra requirement to the definition of $\overline{AE}$ on page 84. This requirement says that if all timestamps in v are smaller than any timestamp in w, then the token with time interval v is consumed before the token with time interval w (even though the values of the two tokens may differ).
More formally, we add the requirement:

$$\forall_{i \in dom(q_{in})} \forall_{j \in dom(s) \backslash dom(q_{in})} \quad \begin{array}{l} place(s(i)) = place(s(j)) \;\Rightarrow \\ time^{min}(s(i)) \leq time^{max}(s(j))) \end{array} \qquad (3.4f)$$

For the event $\overline{e} = \langle \pi_1(e), \overline{s}_1 \restriction f(dom(\pi_2(e))), q\rangle \in \overline{E}$ in the proof of theorem 6, this requirement holds, because requirement (2.4c) holds for the corresponding event e. Therefore, the soundness property, that is theorem 6, also holds for the modified transition system extended with requirement (3.4f). We did not add this requirement in the first place for the sake of simplicity.

Chapter 4

Language and tool

4.1 Motivation

In the previous chapters we have shown that the ITCPN model can be used to model and analyse discrete dynamic systems. However, the practical use of this Petri net model depends to a large extent on the existence of adequate computer tools. Note that this holds for most formal models.
To construct or modify ITCPNs, we need an editor. We also need one or more analysis programs based on the techniques discussed in the previous chapter.
In section 2.4 we defined an ITCPN by a seven tuple (P,V,T,I,O,F,TS). This is a definition in terms of sets, bags and mappings. To create, store, modify and analyse such an ITCPN using a computer, it is necessary to choose a convenient representation comprehensible to a computer program. This representation is called a *language*. In addition a language can have a number of features to facilitate the modelling or analysis of ITCPNs.

We use the specification language *ExSpect* to represent ITCPNs (see Van Hee, Somers and Voorhoeve [53], [55], [56]). The reason we use ExSpect, is the fact that this language is based on a timed coloured Petri net model, called *DES*, which is closely related to the ITCPN model (see Van Hee, Somers and Voorhoeve [53]). In fact, the ITCPN model is a generalization of the DES model in the sense that delays are described by an interval rather than a deterministic value. Therefore, ExSpect can be used for the formal specification of a restricted class of interval timed coloured Petri nets. There is a straightforward relation between this specification language and the ITCPN model.
In this monograph we will use the term 'specification' for descriptions in terms of the language ExSpect. Note that there is a strong relation between the terms 'specification' and 'model'. The term 'model' emphasizes the representation of one or more aspects of a real system. The term 'specification' is used to denote a concise description of the functional behaviour of a system (or software).
As already stated, ExSpect is based on the DES model, a timed coloured Petri net model with deterministic delays. The reason we use the ITCPN model rather

than the DES model is the fact that the ITCPN model is more expressive. Another reason for using the ITCPN model is the fact that interval timing allows for new and powerful analysis techniques. Consider, for example, the concepts refinement and uncolouring defined in chapter 3. It is not possible to define these concepts in terms of the DES model, because in the DES model delays are described by a deterministic value instead of a delay interval.

The reason we pay attention to ExSpect is twofold: (1) we can use ExSpect to specify an ITCPN and (2) we can use the analysis methods described in chapter 3 to analyse ExSpect specifications.
In section 4.2 of this chapter, we will discuss some of the features of ExSpect which facilitate the specification of complex systems.

Based on this language, a software package, also called *ExSpect*, has been developed (see Somers et al. [9]). This software package is composed of a number of tools which have been developed to create, modify and analyse ExSpect specifications. These tools include: a *shell*, a *design interface*, a *type checker*, an *interpreter*, a *runtime interface* and an *analysis tool*.
The author of this monograph participated in the development of two of these tools, viz. the design interface and the analysis tool named *IAT*. The design interface is a graphical editor which can be used to create and modify an ExSpect specification in a user-friendly and graphical manner. The analysis tool can be used to analyse ITCPNs specified by an ExSpect specification. This tool uses the analysis methods described in chapter 3.

Both the ExSpect language and the ExSpect software support the modelling of complex systems in various application domains. However, ExSpect is a general purpose specification language, and therefore, this language is not close to the the professional language used in a specific application domain. This is the reason ExSpect allows for the development of domain specific libraries. These libraries increase the productivity of the modelling process and facilitate the modelling of large and complex systems. The author of this monograph has developed two libraries: one for the modelling of queueing systems (see section 4.5 or [3]) and one for the modelling of complex logistic systems (see section 5.5, [4] or [5]).

4.2 The language

ExSpect (EXecutable SPECification Tool) is a language to describe discrete dynamic systems (see [53], [55], [52], [56], [51], [57], [8], [7]). Moreover, ExSpect is a *constructive* specification language which means that objects (e.g. functions) are specified by a stepwise decomposition into objects that are simple and easily understood. As a result the language is *executable*. Therefore, we can use the ExSpect specification for simulation (or prototyping) purposes.

Like any language ExSpect has a *syntax* and *semantics*. The syntax of a language is a grammar describing the systematic rules of the language. The semantics of ExSpect can be given in terms of the ITCPN model. A part of the semantics of ExSpect is given in Van Hee, Somers and Voorhoeve [51].

ExSpect specifications are stored in *modules*. A module contains a number of definitions. Each definition in a module has a (possibly empty) interface and an implementation. A user of the module only knows about the interface, and the implementation is hidden from the user. This modularization concept is also known as *encapsulation*. Encapsulation hides unnecessary details and if the implementation is changed, a user is not affected as long as the interface is not changed. Note that this is analogous to some of the concepts found in many modern programming languages.

In ExSpect there are four kinds of definitions:

- type definitions
- function definitions
- processor definitions
- system definitions

ExSpect is a typed functional language. Type definitions are used to specify the value set of each place (V_p). Function definitions are used to specify operations on the value of a token (F_t). ExSpect uses the term *processor* instead of transition. A *system* is an aggregate of transitions, places and subsystems. In the remainder of this section we discuss these four kinds of definitions. For a more detailed description of ExSpect, see the ExSpect User Manual [9].

4.2.1 Type definitions

Tokens have a value. This value can be very simple (e.g. a number) or very complex (e.g. a database state). Each place has a *type* which determines which values are allowed for the tokens it contains. To create the suitable types, we need *type definitions*.
The type system of ExSpect consists of some primitive types and a few type constructors. There are five primitive types: `void`, `bool`, `num`, `real` and `str` denoting the 'empty' type, booleans, numerals, reals and strings respectively. The type constructors are set (`$`), Cartesian product (`><`) and mapping (`->`). From a set of types and the type constructors we can form type expressions that symbolize new (composite) types. We can attach names to type expressions, thus defining new types. The following type definitions illustrate this:

```
type weight from real with [x] x >= 0.0;
type volume from real with [x] x >= 0.0;
type manufacturer from str;
type truck from manufacturer >< (weight >< volume);
type truck_id from num;
type fleet_of_trucks from truckid -> truck;
type cargo from weight >< volume;
```

Note that we can add a `with` part for restricting the type.

4.2.2 Function definitions

To specify the value of a produced token, we need *function definitions*. In general, these function definitions are composed out of simpler ones. Our set of basic functions includes all well-known set-theoretical, logical and numerical constants and functions. Some of these basic functions are polymorphic. Because of some 'sugaring' it is possible to write these functions in their usual symbolic infix or 'circumfix' notation. As an example we show two function definitions operating on the types defined above:

```
transportable_by_truck[ c : cargo, t : truck ]
:=
(pi1(c) <= pi1(pi2(t))) and (pi2(c) <= pi2(pi2(t))) : bool;

transportable_by_fleet[ c : cargo, f : fleet_of_trucks ]
:=
if f = {}
   then false        --i.e. there are no trucks left
   else transportable_by_truck(c,pi2(pick(f)))
        or transportable_by_fleet(c,frest(f))
fi : bool;
```

The functions `pi1`, `pi2` (projections), `pick` and `frest` (respectively taking and deleting an element from a mapping) are examples of basic functions.
To define a polymorphic function, we use *type variables*. Consider for example the following function:

```
union[ x : $T, y : $T ]
:=
if x = {}
   then y
   else ins(pick(x),union(rest(x),y))
fi: $T;
```

This function defines the union of two sets having the same type. Since T is a type variable this function can be applied to two sets having an arbitrary type and the result of this function is of the same type. The function is recursive and uses the basic functions `pick`, `ins` and `rest` (respectively taking, inserting and deleting an element from a set).

4.2.3 Processor definitions

In ExSpect we have *processors* and *channels* corresponding to transitions and places respectively. There is also a special kind of channel (place), called *store*, which always contains precisely one token. ExSpect uses these terms, because they seem more natural for people not familiar with Petri nets.

Processor definitions are split in a header and contents part. The header part (sometimes called signature) contains the processor name, its interaction structure and its parameters. The interaction structure is given by (possibly empty) lists of input channels, output channels and stores. The contents part consists of concurrent (conditional) assignments of expressions to output channels and stores. A simple example runs as follows:

```
proc transport_function[in leave:truck,
                        out arrive:truck,
                        val d:time]
:=
arrive <- leave delay d;
```

This processor can be used to model the transport. If there is a token in the input channel `leave`, then the processor is enabled. If the processor `transport_function` remains enabled, it fires (executes) at the time given by the timestamp of the token to be consumed. If it fires, then it produces a token for output channel `arrive` with a value equal to the value of the token consumed. The time between the departure and arrival of a truck is set by a value parameter (`val`) `d`. Note that `delay` is a keyword.

ExSpect has a number of features to make a processor definition highly generic. Besides value parameters it is possible to have function parameters (`fun`). It is also possible to define polymorphic functions. Consider for example the following processor definition:

```
proc p [in a:S, out b:T, val g:$S,
        fun t[x:S] : T, d[x:S] : real]
:=
if a in g
   then b <- t(a) delay d(a)
fi;
```

This processor consumes tokens from the input channel `a`. There is one value parameter (`g`) and there are two function parameters (`t` and `d`). If the value of the token consumed is in the set `g`, then the processor produces a token for output channel `b`. Otherwise, the processor fires without producing a token. The value and delay of the token produced depend on the value of the token consumed. `S` and `T` are type

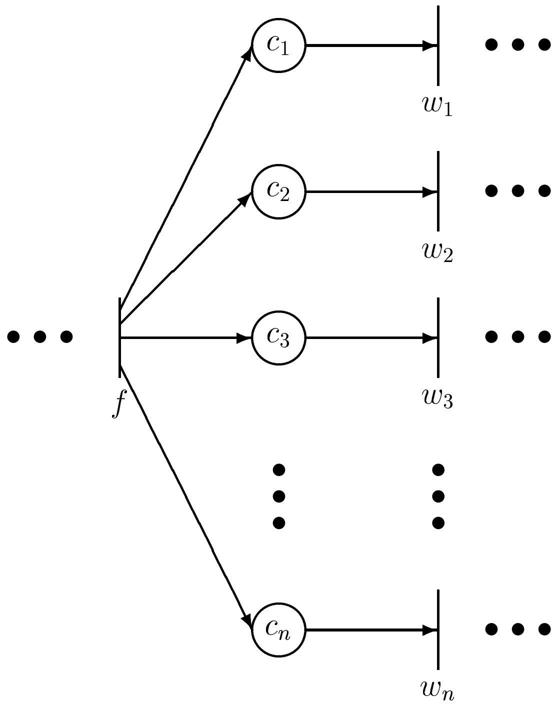


Figure 4.1: A part of a distribution network modelled without preconditions

variables, i.e. the types of the input and output channel are arbitrary. Note that the types of the value and function parameters depend on the actual type of the input and output channel. We will come back to this.

ExSpect release 3.0 (and higher) has been extended with *preconditions* for processors. Consider for example the following processor definition:

```
proc q [in a:real, out b:real | pre a > 0]
:=
b <- ln(a) delay 5.2;
```

This function consumes positive valued tokens from input channel `a`. If this input channel only contains tokens with a value ≤ 0, then q can not fire. In this example we added the precondition, because the logarithm of the value of the consumed token (`ln(a)`) is defined for positive values only.

The concept of preconditions has been added to ExSpect to facilitate the modelling of certain situations that are difficult to specify without preconditions. These situations are found in many logistic systems.
Consider for example a distribution network with one factory and a number of warehouses. Products produced by the factory are transported to one of the warehouses. Figure 4.1 shows a part of this distribution network modelled in terms of channels (places) and processors (transitions) without preconditions. Processor (f) sends a

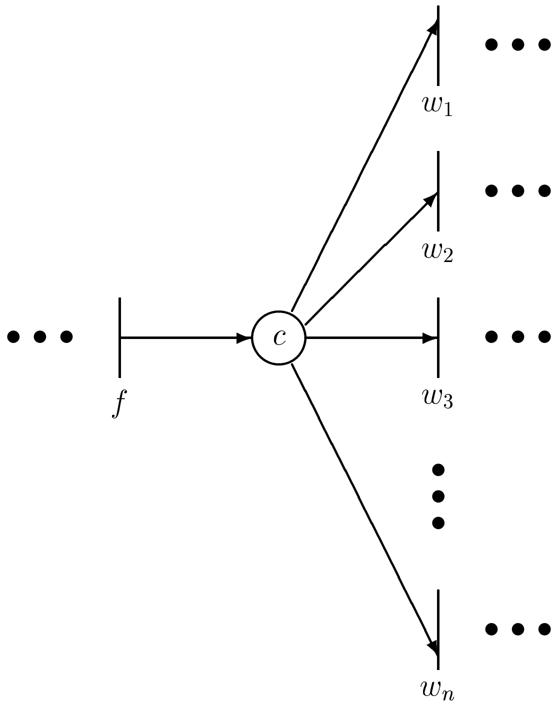


Figure 4.2: A part of a distribution network modelled with preconditions

token to one of the output channels $c_1, c_2, ..c_n$. Each of the processors $w_1, w_2, ..w_n$ represents the acceptance of products in the corresponding warehouse.
For large and more complex distribution networks it is not convenient to model the network like this, because there are a lot of channels and connections. Another drawback is the fact that, every time we decide to add or remove warehouses, we have to change the definition of processor f.
Using preconditions we can avoid these problems. Figure 4.2 shows the part of the distribution network in terms of channels and processors with preconditions. Processor w_1 only accepts products whose destination is the warehouse represented by w_1, etc. Note that there is only one intermediate channel (c) and we can add extra warehouses without changing the definition of f.

In the ITCPN model there is no concept comparable to these preconditions. However, it is possible to transform an ExSpect specification with preconditions into an equivalent ExSpect specification without preconditions (e.g. replace place c by a subnet which tests the preconditions and sends the tokens to a proper transition, if possible). Furthermore, it is possible to extend the ITCPN model and the MTSRT analysis method with preconditions.

4.2.4 System definitions

The main objective of the approach developed in this monograph is to model and analyse large and complex discrete dynamic systems, for example a large distribution

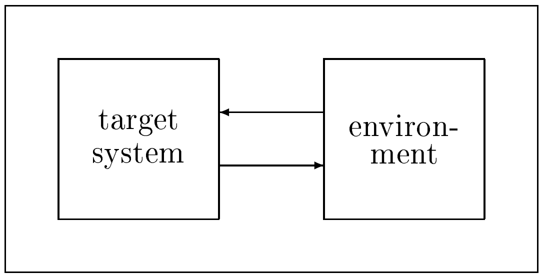


Figure 4.3: The transformation of an open system (the target system) into a closed system (the target system and its environment)

network. Clearly, a specification of such a system in terms of processors (transitions) and channels (places) tends to become too large to handle. This is the reason we added a hierarchy construct, called *system*, to ExSpect.

This construct can be used to structure large and complex systems. The idea is analogous to the hierarchy constructs found in many graphical description languages, e.g. SADT (Marca and McGowan [79]), Yourdon (Yourdon [130]), Statecharts (Harel [48]) and CPN (Jensen [71]).

To clarify this construct, we start with a number of concepts adopted from *systems analysis* (Wetherbe [124]). System analysis is involved with the development of a framework of methods and techniques for evaluating system behaviour. Systems analysis uses an approach which conceptualizes phenomena in terms of wholes consisting of entities or subsystems with the emphasis placed on their interrelationships.

In a general sense, a *system* is a group of *elements* working in an interrelated fashion toward a set of objectives. These elements are the smallest parts to be considered, sometimes referred to as *entities* or *objects*. Each element can be characterized by the *relations* with its environment. Examples of elements are humans, machines, goods or information processing equipment.

The system boundary defines which 'part of the world' is considered and which part is out of scope. It is possible to compose a number of systems into a new system. It is also possible to decompose a system into a number of sub-systems. The latter process can be repeated until we reach the level of elements.

A *closed system* is a system without any interactions with 'some' environment. An *open system* is a system which has a certain (external) interaction structure. Note that it is always possible to transform an open system into a closed system by explicitly modelling its environment. This is expressed in figure 4.3.

Systems are represented by rectangles. We use arrows to denote relations between systems. Nearly all 'real-life' systems are open. Consider for example a human-machine system, i.e. a person interacting with a machine. From a modelling point

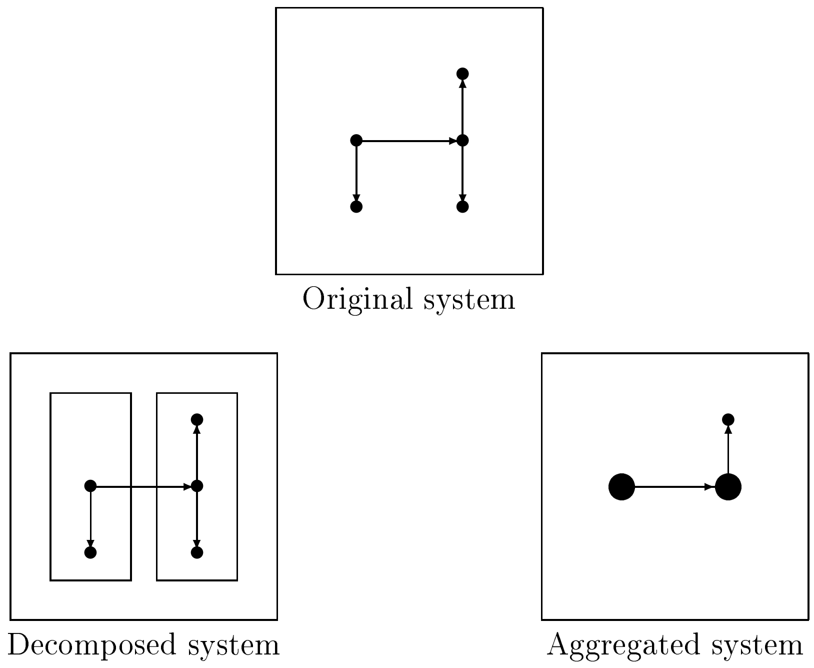


Figure 4.4: The difference between composition and aggregation

of view we can consider such a system as a closed system. This is often useful for analysis. Yet, the human needs food and beverage and the machine needs electricity and maintenance. Note that the environment of a system can only be defined after the system boundary has been defined.

There are a lot of ways to *decompose* (*compose*) a system into (from) a number of smaller subsystems without changing the set of elements (entities). Decomposition is a way to deal with the complexity of systems, because it allows for the consideration of only a small part of the system at the same time. The level of abstraction remains the same, because the set of entities is not changed. If a system X is decomposed into a number of subsystems $X_1, X_2, ..X_n$, then the proper composition of $X_1, X_2, ..X_n$ yields the original system. If we use another set of elementary objects (elements) to model the same system, we speak about *aggregation* (*disaggregation*) rather than composition (decomposition). Using the terminology introduced in section 2.3, we say that the decomposed system is *equivalent* (see definition 9) to the original one, but the aggregated system is merely *similar* (see definition 7) to the original one. An alternative term for aggregation is 'abstraction', i.e. an aggregation step decreases the level of detail. Figure 4.4 shows the difference between composition and aggregation. Note that the decomposed system is equal to the original system. However, the aggregated system is different from the original system, because some of the details are omitted.

In general, a (dis)aggregation or (de)composition step focusses on a specific *aspect*. Typical aspects are (1) functional aspects, (2) spatial aspects and (3) timing aspects. Consider for example a decomposition of a transportation system. We may decompose the system into a number of (geographical disjunct) regions, thus focussing on the spatial aspect. On the other hand, we may decompose the system into two subsystems, one for the transportation of 'fluids' and one for the transportation of 'solids'. In the latter case we focus on a functional aspect.
If we disaggregate a system with respect to the timing aspect, then the dynamical behaviour of a system is modelled more precisely. For example, we model the state of a system every hour instead of every day. In this case, we change the timescale.

Several methods to develop a model (or specification) of a system have been proposed. *Top down* development starts with a model at a high abstraction level, this model is refined by a number of disaggregation steps until the desired level of detail has been reached. To deal with the increasing size and complexity of the model, a disaggregation step often coincides with a decomposition step. *Bottom up* development starts with a model for each of the subsystems. These models are detailed descriptions of some aspect or part of the system, i.e. they have a low abstraction level. These submodels are composed into a model of the entire system. If the overall model becomes too complex, an aggregation step is applied to abstract from some of the details. 'Pure' top down development is often impractical. 'Pure' bottom up development would be a mess. In our opinion, a mixture of top down and bottom up development is the most sensible way to build a model (or specification).

This concludes our introduction to some of the main concepts of systems analysis.

We use a Petri net based approach. The elements (entities) of a system modelled in terms of a Petri net are places (channels) and transitions (processors). The relations between these elements are represented by (graphical) connections.
ExSpect has a hierarchy construct to compose and decompose specifications. This construct is called the *system definition*.
We define a system as an aggregate of processors, connected by channels and stores. A store is a special kind of channel: it always contains precisely one token. A system may also contain other (sub) systems. If a system has no interaction with its environment, then we call it a closed system, otherwise an open system. Open systems communicate with the outside world via input and output channels and stores. Therefore, a system definition consists of a header similar to a processor header and a contents part. A system can have value, function, processor and even system parameters. Thus, it is possible to define generic systems. In this way, a system can be customized or fine-tuned for a specific situation. The contents part is a list of all the objects (processors, systems and local stores and channels) in the system. As an example we show the following system definition:

```
sys ts
:=
channel a:  truck,
channel b:  truck,
transport_function(in a, out b, val 7.25);
```

This is the definition of a closed system with name `ts`, containing two channels and one processor already defined in the previous subsection. Note that there is a clear distinction between the definition of a processor as in:

```
proc transport_function[in leave:truck,
                        out arrive:truck,
                        val d:time]
:=
arrive <- leave delay d;
```

and the *installation* in a system as in:

```
transport_function(in a, out b, val 7.25)
```

Installing a processor means connecting the input and output channels of the processor definition to actual channels inside a system, i.e. to actually use a definition, we must *instantiate* the parameters with actual entities. Note that this is analogous to the separation of a function definition (e.g. `add[x:real,y:real] := x + y : real`) and a function call (e.g. `add(1,2)`).
It is also possible to install systems inside an other system:

```
proc p1 [in i1:S, i2:str, out o:S, fun d[x:S]:real]
:=
o <- i1 delay d(i1);

proc p2 [in i:S, out o1:S, o2:str ]
:=
o1 <- i delay 0.0,
o2 <- 'nil';

sys s1 [in x:S, out y:S, fun d[x:S]:real]
:=
channel free:  str init 'nil',
channel busy:  S,
p1(in x,free, out busy, fun d),
p2(in busy, out y,free);
```

```
wait[x:real]
:=
if x < 0.0 then 0.0 else x fi:real;

sys s2
:=
channel c1:  real,
channel c2:  real,
s1(in c1, out c2, fun wait),
s1(in c1, out c2, fun wait);
```

System `s2` is a closed system (i.e. there are no input and output channels), containing two channels and two subsystems. Note that these subsystems are both installations of the system definition `s1`.
System `s1` has an input channel and an output channel and a function parameter. `S` is a type variable. If we install this system, we can connect the input channel and the output channel to channels of an arbitrary type (as long as they are the same). The two installations of system definition `s1` in `s2`, are both connected to channels of type `real`. Note that the (required) type of the function parameter (`d`) depends on the type of these channels. Both installations of the system definition `s1` in `s2` use the function `wait` with one parameter of type `real`.
The contents of system definition `s1` is formed of two channels and installations of the polymorphic processor definitions `p1` and `p2`.
Figure 4.5 shows a graphical representation of system definition `s2`.

Note that we can replace any system composed of subsystems by a system composed of channels, stores and processors, In other words: it is possible to translate a hierarchical system definition into a behaviourally equivalent non-hierarchical system definition. Consider for example the system definition shown in figure 4.5. If we wipe out the boundaries of the subsystems and rename the internal processors and channels, then we obtain an equivalent non-hierarchical system definition.

For practical applications of ExSpect, the system concept is of the utmost importance. The system concept can be used to structure large specifications. At one level we want to give a simple description of the system (without having to consider all the details). At another level we want to specify a more detailed behaviour. This is supported by a hierarchy construct like our system concept. The system concept also reduces the length of a specification, because we can reuse a system specification (i.e. install a system several times). Polymorphism and several parameter types facilitate the reuse of specifications. Definitions are stored in modules. This way it is possible to hide the implementation of a system definition from the user.

Clearly, the system concept can be used to (de)compose systems. However, in the beginning of this section we also discussed (dis)aggregation, i.e. (dis)abstracting certain aspects. Note that these processes are not supported by a particular con-

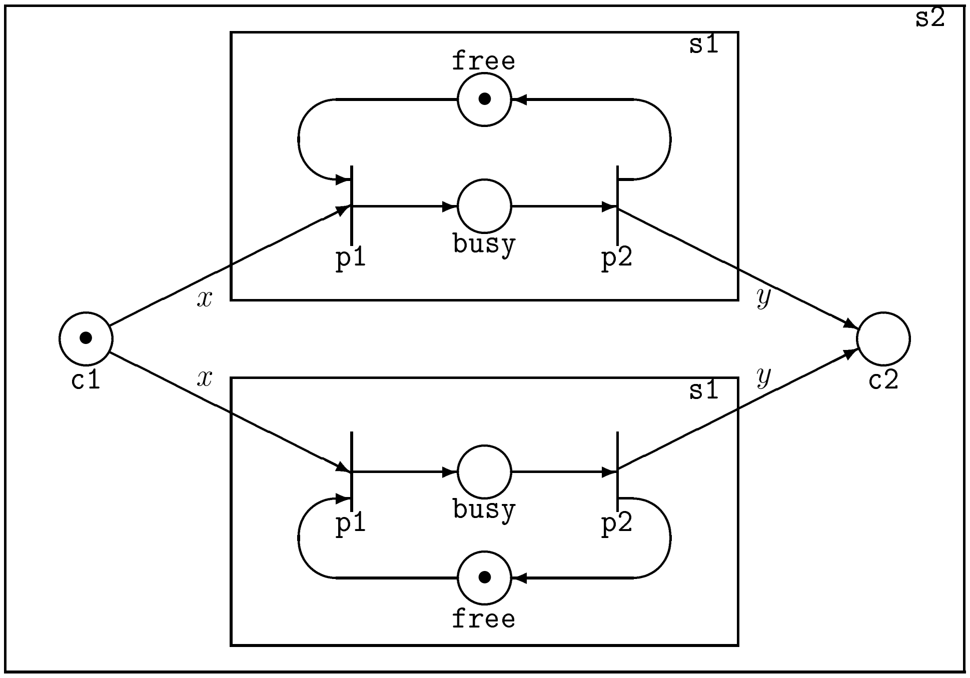


Figure 4.5: System definition s2

cept provided by the ExSpect language. For example, if we decide to disaggregate a system with respect to the timing aspect, then we have to add more detail to various parts of the specification. It is difficult to extend the language ExSpect with concepts which facilitate (dis)aggregation, because changing the abstraction level requires sophisticated transformations affecting varying parts of the specification. Nevertheless, these processes can be supported by tools which facilitate complex modifications of the specification (e.g. replacing a processor by a system).

This concludes our introduction to the language ExSpect. For more information consult the ExSpect User Manual [9] or Van Hee et al. [51]. There are several papers describing the application of ExSpect, see [6], [8], [7], [4] and [5].

As already stated, the reason we pay attention to ExSpect is twofold: (1) we can use ExSpect to specify an ITCPN and (2) we can use the analysis methods described in chapter 3 to analyse ExSpect specifications. The relation between an ExSpect specification and an ITCPN is straightforward except for some details which are discussed by Odijk in [94].

4.3 The software package

To support the language ExSpect, we have developed a software package, also called *ExSpect* (EXecutable SPECification Tool), see Somers et al. [54], [9]. This software package contains a number of computer tools. Basically the set of tools consists of a shell, a graphical editor (design interface), a type checker, an interpreter, a runtime interface and an analysis tool.

For practical applications, the support of computer tools is necessary. There are several reasons which make computer support of crucial importance.

First of all, computer support makes it possible to obtain results which could not have been achieved manually. Most of the analysis techniques mentioned in chapter 3 are unworkable without the aid of an analysis tool. Consider for example the MTSRT method which constructs reachability graphs with thousands of states: it is impracticable to do this manually.

Secondly, computer tools can reduce the number of errors. Calculations by hand are often more error-prone. Furthermore, software can be developed to check the model (specification) for (syntactical) correctness and consistency. This software detects errors like processors without input channels and typing errors. It is also possible to detect deadlocks (traps), siphons, and the absence of certain invariants, etc.

Thirdly, computer support can be used to facilitate the maintenance of models (specifications), because tools can be used to modify a model more easily. With computer support it is often possible to obtain faster results (e.g. modifying or simulating a model).

Finally, there are some additional advantages such as an improved drawing quality of nets, which exceeds the manual capabilities, several on-line 'help' facilities, etc.

As already stated, ExSpect is a set of tools, i.e. a workbench, based on the specification language ExSpect. Figure 4.6 shows the set of tools of ExSpect. These tools are integrated in a *shell*, from which the different tools can be started. The *design interface* is a graphical mouse driven editor, which is used to construct or to modify an ExSpect specification. Such a specification is stored in a source file (module). This source file is checked by the *type checker* for type correctness. If the specification is correct, then the type checker generates an object file, otherwise the errors are reported to the design interface. The *interpreter* uses the object file to simulate the specification. This interpreter is connected to one or more *runtime interfaces*. These interfaces enable one or more users to interact with the running simulation. It is also possible to interact with one or more external programs, for example presentation software. Recently we added the *ITPN Analysis Tool* (IAT) to ExSpect. This tool translates a specification into an ITPN, i.e. an ITCPN whose colour sets have a cardinality of 1, that is analysed using the methods described in chapter 3. The tool also allows for more traditional kinds of analysis, such as the generation of P and T-invariants.

The ExSpect tools have been implemented using C and run under UNIX on SUN hardware. The tools rely heavily on the (simulated) parallelism offered by the UNIX operating system and the graphical capabilities of a SUN workstation.

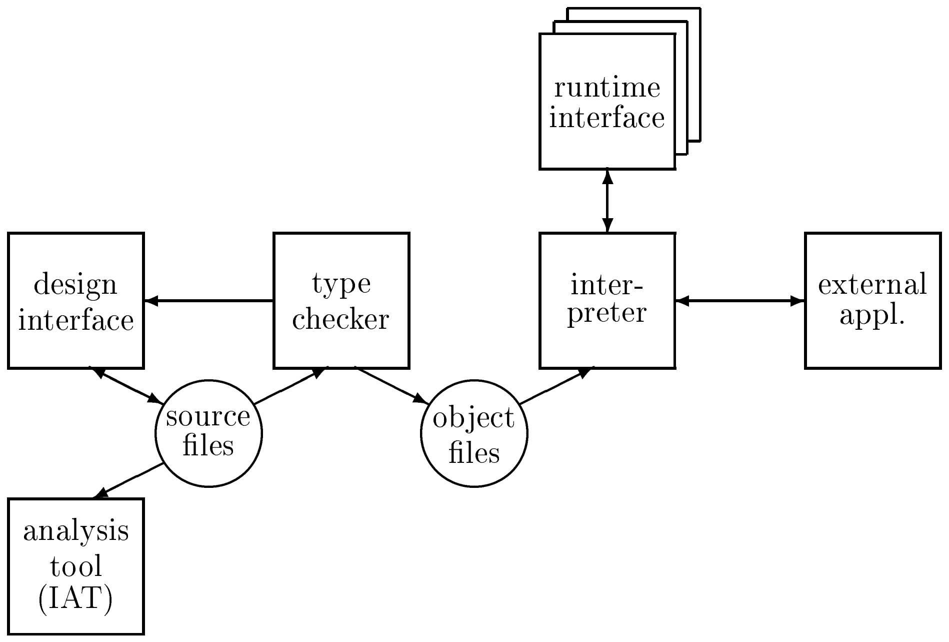


Figure 4.6: The toolset ExSpect

4.3.1 The shell

The primary function of the shell is to integrate the other tools of ExSpect. The shell is used to access other tools, it displays the files in the current directory and it is used to reorganize parts of the file system. Although these things can be done without the aid of the shell, the shell offers more support and is more convenient.
All ExSpect tools have a mouse driven interface and the relevant information can be seen in a number of windows. In the shell you can start one of the other tools by selecting a command from a menu. It is also possible to perform operations on a file by selecting it from a window displaying the current directory.
The shell can be customized by adding personal commands and by specifying your favourite text editor(s).

4.3.2 The design interface

Every module is stored in a separate file, often called source file. To create or modify a module, one can use a text editor such as vi or jove (like when using a programming language). However, such an editor does not meet the requirements set by a language based on a *graphical* formalism. This is the reason we have developed a graphical editor, called the design interface. This tool is window oriented and allows the user to observe, alter and create specifications more easily.
The user is able to edit windows containing graphical representations of systems formed of channels, stores, processors, etc. These windows can be used to create, change or delete graphical objects like channels, stores, processor installations and

system installations. Processor, function and type definitions are edited via easy-to-use forms. At any moment we can print or save (parts of) the specification.

There are a number of settings to modify some of the properties of the tool. This way you can customize the design interface. Most of these settings refer to the graphical part. For example, it is possible to change the default size and shape of the symbols used to represent channels, stores, processors, etc. It is also possible to create new graphical symbols, e.g. an image representing a truck.

Using the design interface instead of a text editor, offers several advantages.

The most important advantage is the fact that given a graphical representation and some additional information, the tool automatically generates a source file. This file also contains some graphical information. At any moment it is possible to switch from the graphical editor to a text editor and vice versa. If the source file does not contain graphical information (or this information is incomplete), then the design interface generates a default screen layout for the system definitions in this file. The user can use the design interface to adjust this layout.

Other advantages are the possibility to do a number of checks and the fact that it is impossible to make certain errors (e.g. connect a processor to a processor).

An additional advantage is the fact that casual and novice users do not have to know the precise syntax of the language. Especially for users who use a rather small subset of the language, the amount of training required is reduced.

The design interface offers the features one nowadays expects from a graphical editor. To conclude we mention two important features.

First of all, the way we handle arcs differs from existing tools in this field (e.g. Design/CPN described in Jensen [71]). A connection (arc) between a processor (or system) and a channel (or store) is considered to be a subordinate to the processor (or system) instead of a separate object. This has the advantage that a connection can be generated implicitly, i.e. the user does not have to bother about drawing a nice arc between two objects. The shape of the generated arc is such that it does not cross objects in the system and the length of the arc is as 'short' as possible. Moreover, if we edit a source file without any graphical information (e.g. a file created with a text editor), then the design interface generates a default layout for each system definition in the file.

Another characteristic of the design interface is the fact that it supports bottom up *and* top down design. We can use already existing definitions by simply typing the name of a definition. This way the user can build a system definition from other, already existing, system definitions (i.e. bottom up). On the other hand, it is also possible to use system definitions (processor definitions) which have not been defined yet. By using a non-existing system (processor) definition we implicitly specify its interface. If we start defining this subsystem (processor), then we 'inherit' its interface (i.e. input and output channels, etc.) based on the way it was used in the suprasystem, i.e. the header of the system (processor) definition is generated automatically. This way it is possible to work top down in a very convenient manner.

4.3.3 The type checker

A source file, created by either a text editor or the design interface, is checked by a tool, called the type checker. This tool checks the type correctness and consistency of the definitions in the source file (also system definitions!). Since ExSpect is a 'strong typed' language, all type checking is done statically. All errors which have been detected are reported in a separate window. If the source file is correct, then the type checker produces an object file (see figure 4.6).
Every source file corresponds to a module. To hide unnecessary details, only a selected set of definitions is visible outside the module. These definitions can be used in other modules which import this module. Each module is checked separately, i.e. type checking is done on a file-by-file basis.

4.3.4 The interpreter and the runtime interface

The object file generated by the type checker can be used to simulate the specification of a system. Simulation is one of the most powerful techniques to analyse a complex system. Simulation is easy to use and flexible in the sense that its application is not limited to a restricted class of systems. An important advantage of simulation is that it helps the experimenter to understand and to gain a feel for the system. In a way, simulation is similar to the debugging of a program, in the sense that it can reveal errors of a (simulation) model.

The task of the interpreter is to simulate a specification. The interpreter is connected to one or more asynchronous user interfaces, called runtime interfaces (see figure 4.6). Each runtime interface is implemented as a separate UNIX process. These interfaces may run on different machines (this is useful for training purposes).
A runtime interface is used to interact with a simulation performed by the interpreter. For example, a runtime interface is able to inspect, add or remove tokens from a channel. All interactions take place via forms. A form has a default layout or it is user defined. This way it is possible to customize the presentation of a running simulation.
It is also possible to connect other external programs to the interpreter. Such a program may be used to present the results in a more convenient way or to analyse some of the data generated by the simulation (e.g. spreadsheets, statistical software).
Unlike many other simulation packages, ExSpect does not support animation. At the moment, the only way to observe the status of a running simulation, is to inspect the channels. This suffices for most simulation purposes, because we are able to present aggregated results in forms. However, for the debugging of a specification, animation seems to be more convenient.

4.3.5 The ITPN Analysis Tool

Although simulation is a very powerful analysis method, it has a number of drawbacks. For example, if the specification contains a lot of non-determinism (e.g. conflicts) or has a highly stochastic behaviour, simulation may be expensive in terms of the computer time necessary to obtain reliable results. Another drawback is the fact that it is not possible to use simulation to *prove* that the system has the desired set of properties. Note that these are the reasons we have developed the analysis methods described in chapter 3.

However, most of the analysis techniques mentioned in chapter 3 are unworkable without the aid of an analysis tool. Consider for example a typical reachability graph, generated by the MTSRT method, with thousands of states. It is impracticable to construct such a graph manually. This is the reason we have developed an analysis tool, called IAT (*ITPN Analysis Tool*).

Because of (software) technical reasons, IAT can only analyse ITCPNs whose colour sets have a cardinality of one. These nets are called interval timed Petri nets (ITPNs). Fortunately, it is possible to *uncolour* an ITCPN, see section 3.5.1 and Odijk [94]. The corresponding uncoloured net can be analysed using the three analysis methods described in chapter 3. The analysis results generated by IAT can be interpreted for the ITCPN that corresponds to the ExSpect specification. In principle it is also possible to *refine* the ITCPN to obtain better results (see section 3.5.2). At the moment this has to be done manually.

```
place p1;
place p2;
place p3 init 2;
place p4;
place p5;
trans t1 in p1, p3 out p2[1.,2.];
trans t2 in p2 out p3, p4[0.,0.5];
trans t3 in p4 out p5[1.,5.];
trans t4 in p4 out p5[2.,8.];
```

Figure 4.7: An IAT source file

To analyse an ExSpect specification, this specification is automatically translated into an IAT source file. This file contains a list of all the places and transitions in the net. Consider for example the file shown in figure 4.7. This example represents a computer system that consists of one CPU and two disks. The structure of the ITCPN for this computer system is given in figure 4.8. In the figure we see that jobs, arriving at the system (p_1), visit the CPU unit before they visit one of the disks. The CPU unit is composed of two parallel processors (initially there are two tokens in p_3). The service time at the CPU is between 1 and 2 seconds. Disk 1 has an access time between 1 and 5 seconds. Disk 2 has an access time between 2 and

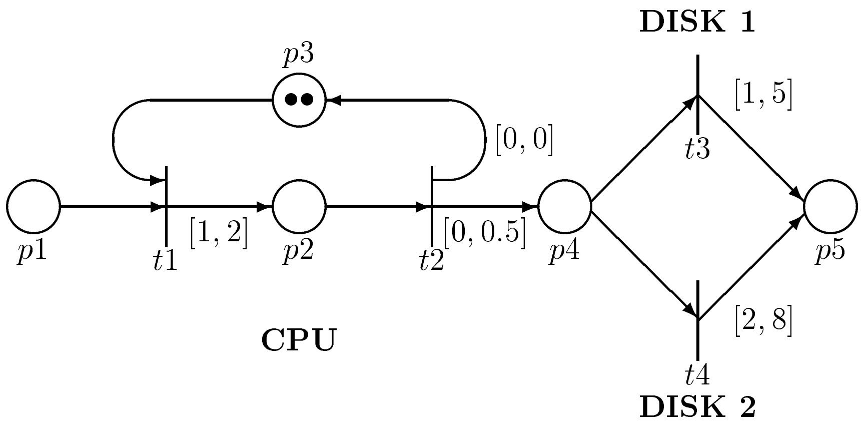


Figure 4.8: An ITCPN for a computer system

8 seconds. Note that the service time at the disk is independent of the load. After a visit to one of the disks, the job leaves the system via p_5.

At the moment IAT supports four kinds of analysis:

- ATCFN
- MTSRT
- PNRT
- calculation of all minimal support place and transition invariants

To calculate the minimal support invariants, we use the algorithm presented by Martinez and Silva in [84] and the modifications described in Colom and Silva [32].

To give an impression of the functionality of the tool, we give some results produced by IAT.
For the net shown in figure 4.8, there are two minimal support place invariants:

$$p2 + p3 = 2$$
$$p1 + p2 + p4 + p5 = 0$$

There are no minimal support transition invariants.
If we tell IAT to compute results based on the ATCFN method, then the window depicted in figure 4.9 appears on the screen. Note that the calculated $\mathcal{EAT}$ and $\mathcal{LAT}$ figures are lower bounds for the actual bounds of the arrival time of the first token, because the net contains a conflict (see section 3.2).

```
STATIC REPORT:


net id : ex______

comments : none___

remarks   name   place   nof_in_trans   nof_out_trans     EAT          LAT        tentative   symEAT   symLAT   init

start     p1 ,    0 ,       0 ,            1 ,        0.000000 ,   0.000000 ,      0 ,        - ,      - ,       0

          p2 ,    1 ,       1 ,            1 ,        1.000000 ,   2.000000 ,      0 ,       t1 ,     t1 ,       0

          p3 ,    2 ,       1 ,            1 ,        0.000000 ,   0.000000 ,      0 ,        - ,      - ,       2

conflict  p4 ,    3 ,       1 ,            2 ,        1.000000 ,   2.500000 ,      0 ,       t2 ,     t2 ,       0

end       p5 ,    4 ,       2 ,            0 ,        2.000000 ,   7.500000 ,      0 ,       t3 ,     t3 ,       0


(end of report)
```

Figure 4.9: Results calculated using the ATCFN method

```
DYNAMIC REPORT:


net id : ex___________

initial state : ex__________

comments : none_________________________


from : 1___ to : 3___


      ||  number   || number of ||
      ||    of     || available ||
place ||  tokens   ||  tokens   ||          1            |          2            |          3
        min   max    min   max      EAT        LAT         EAT        LAT          EAT        LAT
 p1      0     3      0     2    0.000000   0.000000    1.000000   1.000000     1.000000   1.000000
 p2      0     2      0     2    1.000000   2.000000    2.000000     INF          INF        INF
 p3      0     2      0     2    0.000000   0.000000    0.000000   0.000000     0.000000   0.000000
 p4      0     3      0     3    1.000000   2.500000    2.000000     INF        2.000000     INF
 p5      0     3      0     3    2.000000   10.500000   3.000000   11.500000    3.000000   12.500000
(end of report)
```

Figure 4.10: Results calculated using the MTSRT method

Figure 4.10 shows a window which contains some results calculated using the MTSRT method. This window displays information about the arrival times and boundedness given some initial state. There are several ways to calculate the reachability graph. It is possible to select a suitable 'strategy' for this purpose (see Van den Heuvel [61]). Depending on the net and the chosen strategy, IAT is able to generate up to 100.000 states in less than a minute (on a SUN SPARC SLC). Experience tells us that the upper limit of the performance of IAT is more likely to be set by the

available memory than by the processor speed.
An extensive description of IAT is given in Van den Heuvel [61]. Examples of the application of IAT are given in Odijk [94] and in [2].

4.4 Engineering the modelling process

Both the ExSpect language and the ExSpect software, support the modelling and specification of large and complex systems. Although this is true, ExSpect is not a panacea. For example, the powerful constructs provided by ExSpect can be abused to produce unreadable specifications.
As long as the system to be modelled is small there are no problems. However, the modelling process becomes problematic when the system is large or complex. The specification of such a system is often too complex and not transparent enough to comprehend.
To deal with this problem, we propose the development of *domain specific libraries* of *reusable components.* Examples of reusable components are predefined generic system definitions, mathematical function definitions and typical type definitions. The use of these reusable components instead of ad hoc definitions, results in a high-level specification of the system, i.e. the size and complexity of the specification is reduced. There are some other advantages. First of all, reusability is a way to increase the productivity of the modelling process, i.e. it is possible to specify the system in less time. Secondly, domain specific libraries of reusable components can be used to capture knowledge. When making a reusable component, some domain knowledge is acquired by the modeller. In a way, this knowledge is stored in the components. Reusing these components facilitates the diffusion of this knowledge.

A domain specific library is composed of a number of modules containing all sorts of definitions. These predefined definitions are called *components* or *building blocks.* The term 'building block' expresses the fact that we are able to combine system definitions graphically. This is also the reason we sometimes use the term *toolbox* instead of domain specific library.

Basically, there are two ways to reuse these components.
The most easy way to reuse predefined components, is to use them *without any modifications.* Consider for example the use of a library containing standard mathematical functions. Another example is the use of predefined subsystems like a 'generator', 'duplicator' and 'absorber'. If present, parameters can be used to customize the component. Suitable parameters make a component *generic*, i.e. it can be applied in many situations. To use components in this manner, it suffices to know the header of the definition (i.e. it is not necessary to know the internal structure).
Another way to reuse specifications, is to modify parts of already existing components. This kind of reuse poses a number of problems. To modify existing definitions, the user needs information about the exact (internal) behaviour of such a component. Without a full understanding of the operation of the component, this kind of

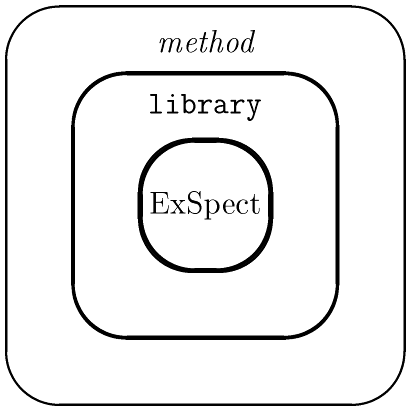


Figure 4.11: The result of domain analysis

reuse is dangerous and likely to cause errors. Therefore, we dissuade this kind of reuse by novice users.

Our main interest in domain specific libraries stems from the realization that reuse supports and speeds up the modelling process. We aim at a '80/20'-situation, where 80 percent of the components needed are already available in standard libraries and take up only 20 percent of your time. But the 20 percent you have to create yourself take up 80 percent of your time. This 80 percent includes the time used to modify existing components.

Clearly, this is an ambitious goal. We think this situation is feasible in various domains (supported by a domain specific library), because ExSpect has a number of constructs that allow for the development of highly generic components, which are easy to use. To motivate this statement, we list some of the constructs which have been described in section 4.2.
ExSpect supports composition and decomposition by a hierarchy construct, called system. The system concept offers the possibility to use generic building blocks that can be combined graphically.
The module concept can be used to hide unnecessary details (encapsulation).
Polymorphism by type variables is very powerful, since it allows for the development of components which are (partially) independent of the actual type of an instantiated parameter. For example, it is possible to define functions that can be applied to any kind of set (e.g. the definition of `union`).
Processor and system definitions can have several parameters (a system definition can even have processor and system parameters). This way it is possible to develop highly generic components, which can be customized for a specific application.

To develop powerful toolboxes, we have to do some *domain analysis*. In this monograph we consider domain analysis as an activity prior to the actual modelling of a

particular system and whose output supports the modelling of any system in this domain. Domain analysis tries to generalize situations rather than focus on a particular system. The result of domain analysis is a library which transcends a specific application.
In a way, ExSpect and such a library make up a *domain specific language.* This way it is possible to create a language which is close to the user's professional language. Nevertheless, for some application domains this is not sufficient, because the complexity of the problem requires a systematic approach. To support the use of a library in such an application domain, we have to supply a *method.* This is expressed in figure 4.11. Such a method is a collection of rules and guidelines to support the modelling process in a systematic manner.

A library is formed of a number of modules. Some of these modules are also part of other libraries. At the moment there are five standard modules:

basic elementary mathematical and logical functions, like set insertion, etc.

utils more mathematical and logical functions

stat statistical functions, e.g. a function to generate random numbers

adt operations on abstract data types like arrays, lists and bags

qn components that can be used to model queueing networks

We have used ExSpect to model all sorts of systems: queueing systems, information systems, protocols, production systems, etc. However, our main interest is in the field of logistics. In chapter 5 we describe a library that has been developed for this domain. In this chapter we also propose a method to facilitate the modelling of logistic systems.
To illustrate the use of a domain specific library, we sketch the *QNM library,* which is composed of only one module: the `qn` module (see Van der Aalst [3]). Since the components in this module are self-explanatory for users familiar with queueing networks, we restrain ourselves from presenting a method.

4.5 A library: QNM

In the last twenty years, queueing networks have become popular in the field of performance analysis of computer systems, communication networks and production systems. A common feature of all these systems is the fact that there is a limited resource which must be shared among a number of competing customers that require service. Examples of typical shared resources are CPUs, memory, I/O devices, transport aids and machines. Since these resources are limited, customers may have to wait. These waiting customers form a queue in front of the shared resource. This is the reason these systems are called *queueing systems.* In other words: a queueing system is a network of queues and servers containing a number of customers (clients) circulating in the network.

There exist two approaches for the analysis of queueing networks.
The most flexible and easy-to-use method is simulation. Simulation can be applied in many situations and, by nature, it provides the opportunity to model and analyse systems which are mathematically intractable.
Simulation is not the only way to analyse queueing systems; 'pure' queueing systems also allow for analytical methods. In fact the main reason for which queueing networks have become so popular is due to the *product form* solution property that holds for a fairly large class of queueing networks (see Baskett et al. [13]). Nevertheless, several practically important features, like synchronization, blocking and the splitting of customers can usually not be modelled in such a way that the model still has the product form solution (see Ajmone Marsan [83]). For non-product-form queueing networks there are approximative methods of analysis available, but these are not generally applicable and require an expert consultant. Therefore, for a more detailed analysis of queueing systems simulation is practically unavoidable.

We propose a hybrid approach. This approach is based on the *Queueing Network Module* (QNM), a library containing one module (`qn`). This module contains a number of building blocks, like a generator, a server, a queue, etc. These building blocks allow for the modelling of a fairly large class of queueing networks, in a graphical manner. The design interface automatically generates a simulation model allowing for all sorts of measurements.
Under certain conditions, it is possible to translate such a model into a BCMP network (see Baskett et al. [13]). Such a network can be analysed using standard analytical techniques. If these conditions are violated, then the simulation results are still useful; they can be used to obtain parameters for an approximated BCMP network or to compare them with the results of an analytical technique.
See [3] for more information about the relation between QNM and BCMP networks.

One can think of QNM as an *interface* on top of ExSpect. This interface prevents the user familiar with queueing networks from having to learn a new formalism. It fully utilizes the features of the language ExSpect such as polymorphism, value and function parameters, hierarchical modelling and encapsulation.

The `qn` module contains definitions of the following building blocks:

generator The `generator` component takes care of the generation of new customers.

server The `server` component satisfies the needs or requirements of arriving customers. Most servers have a limited capacity, i.e. the number of customers being served at the same time is restricted.

queue When a server is too busy to serve incoming customers, these customers have to wait for their turn. Upon completion of a service, the queue selects the customer that must be serviced next, according to some queueing discipline. The `queue` component takes care of the buffering and selection of waiting customers.

assign Sometimes we want to model one queue in front of a number of (possibly non-identical) servers. To do this we have to use the `assign` building block.

selector The `selector` component takes care of the routing of customers.

assemble The `assemble` building block is used to synchronize two queues.

term Customers arriving at the `term` component leave the queueing network.

Each component has its own graphical symbol. Figure 4.12 shows the symbols used by the design interface to picture a queueing network. A complete description of these components is given in [3]. In this section we focus on two components: the `server` and `queue` system.

Server

The customers in the network travel from server to server until they leave the system. At each server they offer a certain amount of work (the *workload*) and they wait until the server has completed the service. One can think of a server as a service point or a workstation. A server is always connected to a queue (sometimes indirectly via an `assemble` and/or an `assign` system). If the server is free and there are customers waiting to be processed by this server, then the `server` system starts serving one of the customers. The service time is given by a probability distribution which may depend upon the value of the customer. One can also use the `server` system to model a number of identical parallel servers or an infinite server (i.e. a station with an infinite number of servers).
The header of the `server` system looks as follows:

```
sys server[
           in i:S,
           out o:T, sig:signal,
           val name:str,
               seed:real,
               nofservers:num,
           fun servicetime[x:S,r:real]:real,
               transform[x:S]:T
          ];
```

The `server` system is polymorphic, because `S` and `T` are type variables. Input channel `i` and output channel `o` are used to model the arrival and departure of customers. There is one other output channel, called `sig`, that is used to inform the

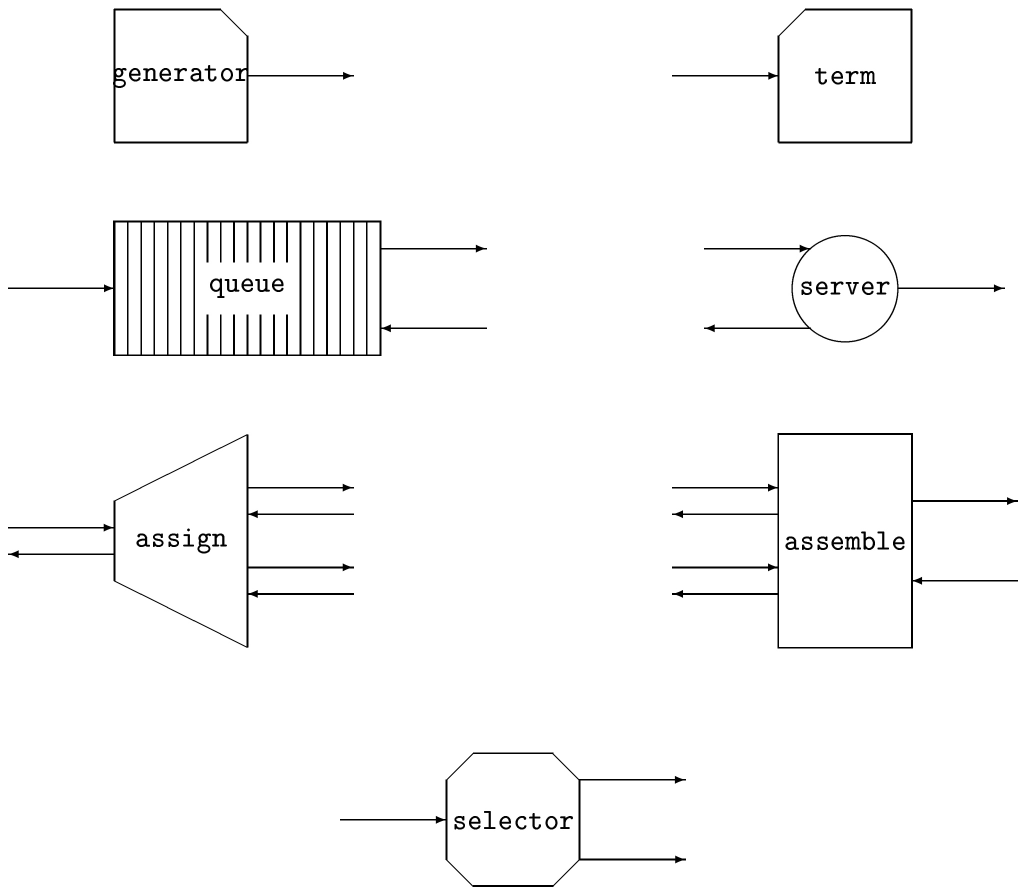


Figure 4.12: The symbols used by the design interface

preceding `queue` system (or `assign` or `assemble` system) that the server is ready to process another customer. The output channel `sig` is of type `signal`, a predefined type with only one element, also called `signal`. The value parameter `name` is used to specify the name of the server and `seed` is used to set the random generator. The number of parallel servers inside the `server` system can be specified via `nofservers`, this value parameter is set to `INF` to model an infinite server. The service time of a customer is given by the function parameter `servicetime` and may depend upon the value of the customer (`x`) and a random number (`r`). Note that the service time may be fixed, calculated by an expression or random with a particular probability distribution. Since a service can change the attributes of a customer, a function parameter, called `transform`, is supplied. The input of this function is the value of the arriving customer (`x`), the output is the value of the processed customer. Note that the resulting type of this function and the type of the output channel `o` have to 'match'.

Queue

A queue is used to store the customers waiting to be served. The order in which the customers leave the queue is defined by the *service discipline*, for example FIFO (first-in-first-out), LIFO (last-in-first-out) or SIRO (select-in-random-order). A `queue` system is always followed by a `server`, `assemble` or `assign` system.

```
sys queue[
            in i:T, sig:signal,
            out o:T,
            val name:str,
                seed:real,
            fun discipline[n:num,x:T,r:real]:real
           ];
```

Input channel `i` and output channel `o` are used to model the arrival and departure of customers in a queue. Note that `T` is a type variable. The input channel `i` receives customers from `generator`, `server` and `selector` systems. There is also an input channel called `sig` used by a `server`, `assemble` or `assign` system to send a message to tell the queue that it is ready to accept new customers. If the queue contains customers and there is a token in the input channel `sig`, then a customer is selected and sent to the `server`, `assemble` or `assign` system. The name of the queue is specified by a value parameter called `name`. The function parameter `discipline` is used to specify the service discipline, which may depend upon the arrivalnumber (`n`), the value of the customer (`x`) and a random number (`r`). The function returns a real value for every queued customer, i.e. `discipline` assigns a *weight* to every waiting customer. The queue always selects the customer with the *highest* weight to leave the queue. Note that this way it is possible to specify various service disciplines, e.g. FIFO, LIFO, SIRO, priority scheduling, etc.

To illustrate the use of the QNM library, we present a small example. In this example the QNM building blocks are used to model a jobshop producing rolled products.
The jobshop receives iron bars from a blast-furnace plant. These bars are transformed into steel plates using rolling mills to flatten the iron bars, and cutting machines. This transformation process takes a number of steps. The sequence of operations transforming an iron bar into a finished product is called a *job*. Since the QNM building blocks are polymorphic, we have to specify a type describing the attributes of a customer. If the user does not want to bother about this, (s)he can use a predefined type, called `client` (see [3]). In most cases this type is convenient. However, in this example we define our own type (`job`):

```
type product from str;
type operation from str;
type date from real;
type duration from real;
```

job					
product	operation seq.			date	date
	num	operation	duration		
'AA34234'	1	'weldingA436'	1.2	11.28	12.5
	2	'weldingB476'	1.6		
	3	'cuttingC132'	0.2		
	4	'weldingB462'	2.0		
	5	'cuttingC773'	0.4		

Table 4.1: A value of type `job`

```
type job from product ><                              -- product code
              (num -> (operation >< duration)) >< -- operation seq.
              date ><                                 -- start date
              date;                                   -- due date
```

A job has four attributes, viz. a product code, a sequence of operations, a start date and a due date. The product code specifies the type of product that has to be produced. The sequence of operations represents the (ordered) set of operations that have to be performed before the product is ready. For every operation we specify the estimated processing time. The start date is the date the job has been released. The due date represents the date the product has to be available. A value of type `job` is shown table 4.1.

The jobshop described in this example has two rolling machines, a 'two-high rolling mill' and a 'universal rolling mill'. For convenience, we will call these machines A and B. Every rolling operation is assigned to precisely one rolling mill, i.e. operations are machine specific. There is also one universal cutting machine (machine C). A rolling operation is always followed by a lubrication operation, performed by machine D. This machine applies a lubricant to make the product smooth.

Figure 4.13 shows the corresponding queueing network in terms of the QNM building blocks. Every `server` system corresponds to a machine. The service time distribution at a server depends on the type of operation. The selector systems take care of the routing of jobs. The service discipline of the cutting machine is first-in-first-out (FIFO). The two rolling mills have a queueing discipline to minimize the lateness of jobs. This service discipline is called `EarliestDueDate`, i.e. jobs with the earliest due date are selected first.

```
EarliestDueDate[ n : num, x : job, r : real ]
:= - pi2(x) : real;
```

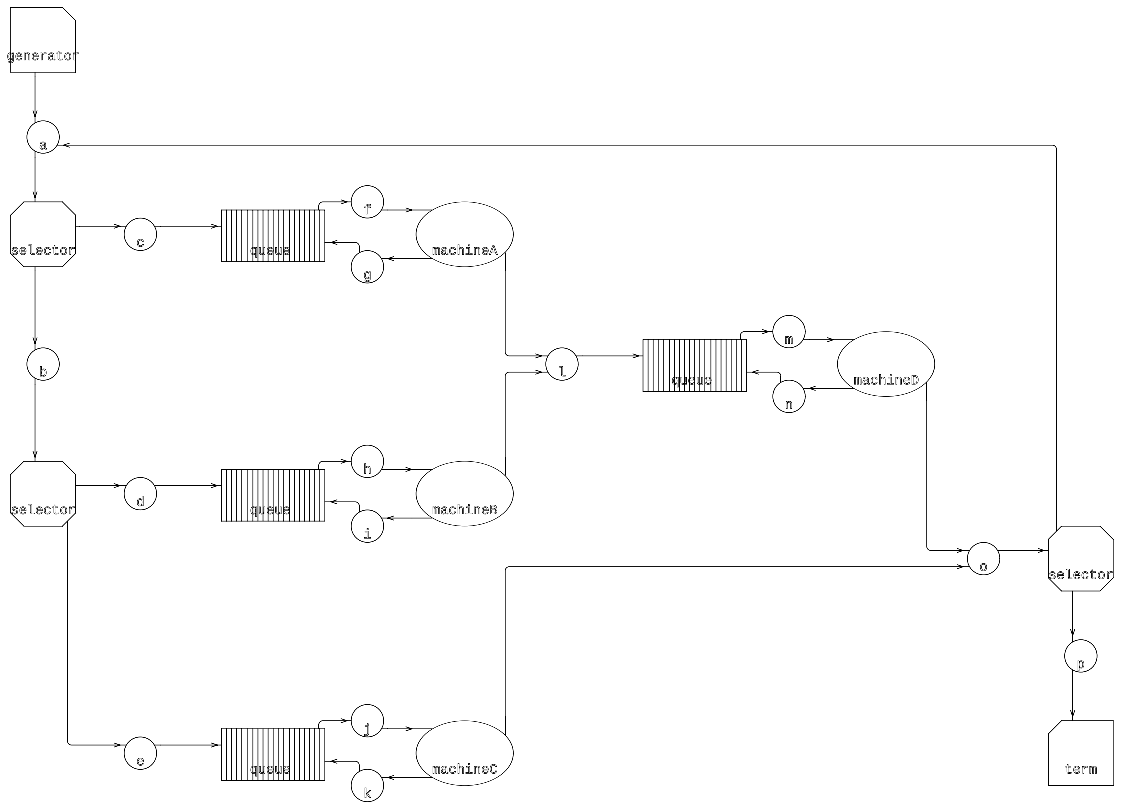


Figure 4.13: A jobshop modelled with QNM-ExSpect

Note that jobs are selected in descending order of their due date (`pi2(x)`). Machine D uses priority scheduling, jobs are discriminated by the machine they come from. Products coming from machine A have priority over products coming from machine B, because they tend to be voluminous. Products coming from the same machine are serviced in order of their arrival (FIFO).

The graphical representation (figure 4.13) of the jobshop was created with de design interface of ExSpect. The structure of the model is defined in a completely graphical way. This only takes a few minutes. To feed the model with parameters (distributions, queueing disciplines, etc.) also takes a few minutes. Then the model is ready to be simulated. The runtime interface allows the user to observe a running simulation. During the simulation the runtime interface reports several measurements (waiting times, queue lengths, etc.). It is also possible to export data to a statistical package or presentation software.

For an important class of queueing systems we have created a ‘100/100’-situation, i.e. all components (100 percent) needed are already available in the `qn` module, and therefore, the usage of these components takes up 100 percent of your time. This class includes queueing systems subject to phenomena such as synchronization,

blocking and the splitting of customers. Features which cannot be modelled with the building blocks described in this section, are 'preemption' (i.e. a service is interrupted) and service times depending on the queue length.
The module `qn` fully exploits the graphical capabilities of ExSpect. Our approach combines the advantages of a *simulation package* (focused on a limited field of applications) and a *simulation language* (flexible, but not easy to use). Another advantage is the possibility to create your own building blocks using a hierarchy construct. This is an important improvement compared to other graphical simulation tools. A much more detailed description of QNM can be found in [3].

Chapter 5

Modelling logistic systems

5.1 Introduction

Modern organizations are required to offer a wide variety of products, in less time than previously and at competitive prices. To meet these requirements such an organization has to devote a lot of energy to a continuous improvement of its logistic performance. To improve the overall logistic performance, it is necessary to investigate how the logistic components contribute to the logistic performance of the organization as a whole. Clearly, this is a complicated task. In this chapter we focus on means to support this task. In particular, we investigate which role the theory, tools and methods described in the previous chapters can play in the area.

Our contribution to the solution of problems related to the modelling and analysis of complex logistic systems is threefold:

1. An answer to the question: 'Why Petri nets?'. To motivate the fact that we use a Petri net based approach, we will show that, in general, a logistic process can be represented by a Petri net in a very natural manner (e.g. goods and capacity resources are represented by tokens, buffers, storage space and media are represented by places, and operations are represented by transitions). We also show how to model typical logistic processes in terms of timed coloured Petri nets. Other reasons to use Petri nets are the graphical nature, the firm mathematical foundation, the analysis methods and the availability of computer support. Furthermore, we will compare our approach with more conventional approaches used to model and/or analyse logistic systems.

2. Another contribution of this research lies in the construction of a 'systems view' of logistics. Based on a taxonomy of the flows in a logistic system, we describe a systematic approach to the modelling of logistic systems. This approach is used to structure the field of logistics, e.g. we identify typical control structures.

3. Finally, we have developed an ExSpect library of logistic components, based on our systems view of logistics. These components are generally applicable

and therefore they can be used in a variety of logistic applications. In a way, ExSpect and this library make up a domain specific language, close to the user's professional language.

In the previous chapters we discussed concepts, techniques and tools to model and analyse discrete dynamic systems. Therefore, we restrict ourselves to logistic systems which are *discrete*, i.e. the flows of goods, materials, capacity resources, information and control are composed of identifiable entities.

The approach presented in this chapter is characterized by a number of salient features.
First of all, our approach provides an integrated perspective for various logistic flows, i.e. flows of goods, capacity resources, information and control are modelled using the same concepts. Moreover, we focus on the main logistic functions (e.g. transport, demand, supply, production and stock holding) in a unifying way. This is possible, because we restrict ourselves to the functional behaviour of the system and we ignore aspects like administration, safety, personnel, etc. If convenient, we also abstract from the physical reality, i.e. we are not interested in the actual layout of a logistic system, mechanical aspects, communication protocols, etc.
Secondly, our approach is characterized by the fact that during the modelling process the user is not shackled by the techniques that are going to be used to analyse the model. Many techniques used in operations research, enforce implicit modelling decisions, i.e. the problem statement is simplified to allow for analytical solutions. Furthermore, the analysis techniques to be used depend on the questions that have to be answered, i.e. sometimes different types of analytical models (solvers) are used to answer different questions within the same situation. Therefore, in [122] and [123], Wessels advocates the use of a 'solver-independent' medium for the modelling of the system, e.g. Petri nets. Modelling in terms of timed coloured Petri nets is characterized by a high degree of freedom. Moreover, timed coloured Petri nets allow for various kinds of analysis, see chapter 3. Therefore, we can use one model to analyse the system using different kinds of analysis.

To clarify the problems we are dealing with, we start with a short introduction to logistics. The rest of this chapter deals with our approach, based on timed coloured Petri nets, concepts from systems analysis and knowledge from logistics as an application domain.

5.2 Logistics

This section provides a short introduction to the nature and purpose of logistics, intended for readers not familiar with logistics.
A logistic process consists of the flow of goods and services and the monitoring and control of these flows. Typical activities include: transportation, inventory management, order processing, warehousing, distribution and production. *Logistics*

management is concerned with the development of functions to support these activities. A simplified definition of logistics is: "The process of having the right quantity of the right item in the right place at the right time" (Hutchinson [68]).

The period of the early 1950s through the 1960s represents the takeoff period for logistic theory and practice. Prior to this time, the field was in a state of dormancy (except for military logistics). Business fragmented their management of the key logistic activities, i.e. there was no integration of the logistic activities. Some reasons for the increased interest in logistics are: a squeeze on profits during this period because of the economic climate and the increased variety in the goods demanded by the consumers. The recession in the early 1970s stimulated a change in priorities from the production of products to the service of demand. There was a shift from a 'sellers market' to a 'buyers market', which forced companies to offer a diversity of products in a swiftly evolving market. Companies were forced to react swiftly upon changes in the market and to deliver an increasing variety of high-quality products within tight terms of delivery. To meet these requirements, companies had to improve the control of their logistic activities. During these years there was a trend towards the integration of the logistic activities to improve efficiency and to reduce costs. This trend still exists and is stimulated by progress in computer technology allowing for more complex calculations.

Logistics management often has to deal with conflicting interests within the same enterprise. Consider, for example, the stock levels inside a company. The marketing and production departments like to have high stock levels to be able to sell from stock and to produce in large batches. The financial department, however, likes to have minimal inventories to reduce interest costs and the costs of loss of inventory due to deterioration or getting out of date. To avoid sub-optimal solutions the *total cost* concept was developed. The total cost concept reflects the recognition that conflicting cost patterns should be examined collectively. For example, when choosing the mode of transportation, the total cost concept would encourage us to consider the impact of the decision on the firms inventory.

The two main objectives of logistics management are a reduction of the overall logistic costs and an improvement of the service provided to the customers of the firm. These objectives have to be balanced at optimum (depending on the branch of industry and the firm's competitive situation). Therefore, we divide the *logistic performance* into the *internal logistic performance* and the *external logistic performance*. The external logistic performance is often called the *customer service* capability. Elements of customer service are: availability, average delivery time, deviation of delivery time, flexibility and quality. The internal logistic performance refers to the efficiency of a logistic system to maintain a certain customer service level. Elements of the internal logistic performance are: stock levels, number of transports, number of setups, required supplier performance, average lead time and handling costs. Note that the internal logistic performance is directly related to the

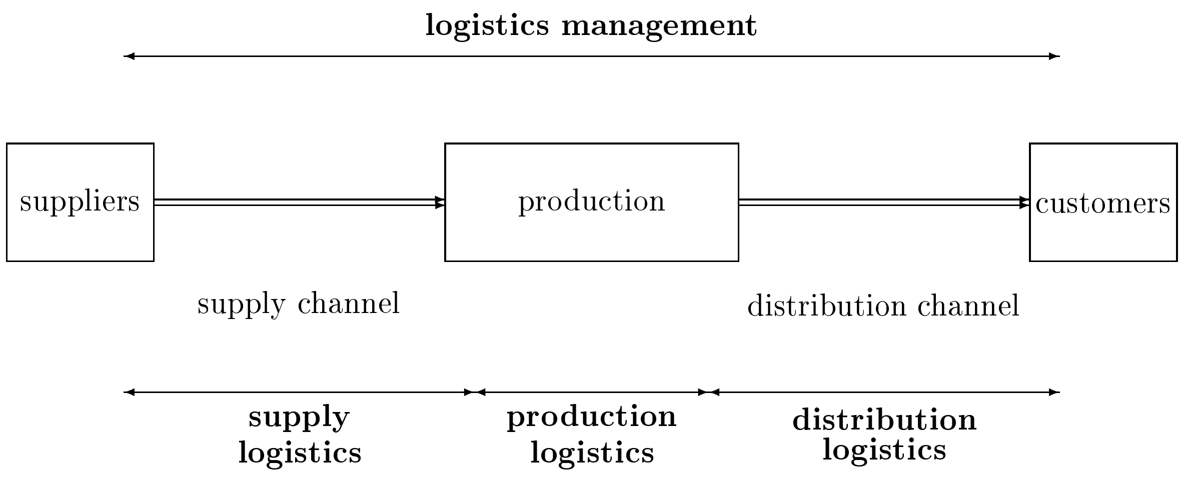


Figure 5.1: A schematic view of a logistic chain

total cost of the logistic process.

The logistic performance is often measured using (key) *performance indicators*. A performance indicator quantifies a specific aspect of the logistic performance. Examples of performance indicators are the percentage of deliveries exceeding their due date, the percentage of backorders and the average stock levels.

We often call the channel with goods flows and information from supplier to consumer the *logistic chain*. The flow of materials and products in a logistic chain proceeds through a series of consecutive locations as it moves from origin to the final destination. This flow of goods has to be controlled, logistics management (or *business logistics*) takes care of the overall coordination of the logistic chain. For discussion, logistic operations are divided into three categories: (1) supply logistics, (2) production logistics and (3) distribution logistics.

The task of *supply logistics* is to satisfy the needs of an operating system, such as a manufacturing production line or a warehouse, i.e. it controls the inbound flow of materials. Supply logistics manages the part of the logistic chain called *supply channel*. Typical activities in the supply channel are: acquisition of materials, materials handling, transportation of supplies to the plant and the maintenance of the inventories at the plant.

Production logistics controls the flow of semifinished components, i.e. the flow of goods between the stages of manufacturing. The objective of production logistics is to control the goods flow such that the products are produced at the right time in the right quantity given the operational (capacity) constraints of the production process.

Distribution logistics is concerned with the movement of products to the customers. It deals with the transport, storage and service of goods that need no further processing within the firm. These finished goods are stored in a central warehouse, a field warehouse or shipped directly to the customer. The main objective of distribution logistics is to provide the availability of the product to the customers as and when they desire it and at minimal costs. The part of the channel controlled by distribution logistics is called the *distribution channel.* Typical decisions are: where to locate inventories (and how large) and the mode of transportation.

Figure 5.1 shows a logistic chain. Several authors (e.g. Bowersox [24]) use the terms *materials management* and *physical distribution* instead of supply logistics and distribution logistics respectively. However, these terms are also used to describe specific aspects of logistics management.

Note that the total logistic chain (from raw material to the consumption of end-products) often stretches out over a number of different enterprises. If we consider a number of companies at the same time, we talk about *interorganizational logistics.* The role of a company depends upon the scope of the logistic chain we want to consider. From a manufacturers point of view a retailer is a consumer. From a retailers point of view a customer is a consumer and the manufacturer is a supplier.

The management of the set of logistic processes can be decomposed into a *hierarchy.* Figure 5.2 shows a typical control hierarchy for the field of logistics. Most authors distinguish three categories of decisions: (1) strategical decisions, (2) tactical decisions and (3) operational decisions (see Anthony [11]).

Decision making at a strategical level is the process of establishing corporate goals and organizational objectives. Expanding marketing activities into a new geographical territory, introducing new products and building plants are typical strategical decisions. An interesting strategical question in the field of physical distribution is: 'Do we need regional warehouses?'. The time span of such a decision is long (several years) and the impact is high.

Tactical decisions are made to select the methods to achieve organizational objectives. At this level we are concerned with aggregate production rates and aggregate inventory levels. Examples of tactical decisions are the decision to buy an extra machine and the construction of the Master Production Schedule (MPS).

Operational decisions are made to control the manufacturing and logistic processes from day to day. These decisions are at a detailed level and the impact on the entire logistic chain is low. However, the frequency of these decisions is very high. Examples are detailed scheduling, dispatching and routing.

Note that each level has a different function and operates on a different *time scale.* Time scales may range from years and months to minutes and seconds on the shop floor. Even within the same level, these time scales may vary considerably.

In this chapter we propose an approach to the modelling and analysis of logistic systems, based on the concepts introduced in the previous chapters. The total cost

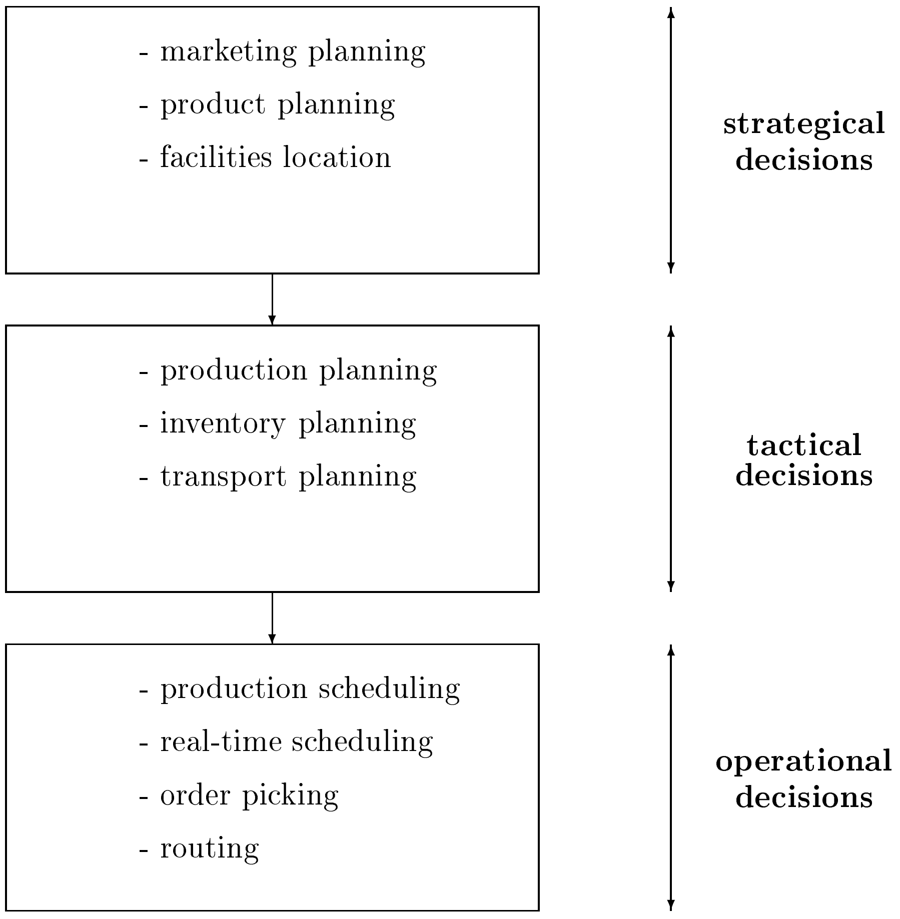


Figure 5.2: A control hierarchy

concept and the importance of a good customer service force us to consider the entire logistic chain. The observation that there are at least three levels of control (strategical, tactical and operational) shows the need for an integrated framework, i.e. a framework which is able to deal with logistic problems at various abstraction levels.
We propose a framework based on a timed coloured Petri net model. This framework is made up of the software package ExSpect, a logistic library and a method which provides guidelines for modelling with this library.

5.3 Why Petri nets ?

To realize the objectives, set out in the previous section, we use a framework based on a timed coloured Petri net model. In this section we will motivate our choice to use a Petri net based approach. We will show that Petri nets allow for a natural representation of discrete logistic processes. Note that there are several other reasons to use Petri nets, e.g. the graphical nature, the firm mathematical foundation, the analysis methods and the availability of computer support. We also discuss some of the features of ExSpect in the light of logistics.

Logistic systems take care of the flow and storage of raw materials, in-process inventory and finished goods. Since stored goods can be seen as flowing goods with speed zero, the main objective of logistics is to control the flow of goods. We suppose that these goods are discrete, i.e. it is possible to identify single products. Examples of non-discrete logistic systems are the production and transportation via pipelines of liquids and gasses. In most cases, however, it is possible to model a continuous process in a discrete way.

Besides the flow of goods, a logistic system comprises a diversity of information flows (e.g. control flow, orders, requests) and means (e.g. machines, tools, manpower). Hence, we require a framework which allows for the modelling of these flows in a unifying way.

A logistic system is *distributed* over a number of sites. For example: demand, supply, production and storage often occur at different geographical locations. This implies that various processes happen at the same time, i.e. in *parallel*. To model these processes it is convenient to have a graphical formalism which expresses the distributed aspect of a logistic system.

Since a logistic system comprises processes happening in parallel, it is necessary to be able to model *synchronization*. Synchronization is also induced by assembly (an operation has to wait for specific goods) and control (an operation has to wait for the proper command), etc.

In the remainder of this section we will show that Petri nets, extended with time and colour, come up to the requirements just stated.

First of all, Petri nets are well suited to *model many different logistic flows in a unifying way*. Modelling the flow of goods, means, information, etc., by tokens seems to be very natural. A place either represents a medium through which something is sent or some storage space (i.e. a buffer). The fact that flows are represented graphically is a very important quality, since it makes the overall structure comprehensible and supports the communication between people having different backgrounds.

We focus on discrete processes, i.e. products, pieces of information, etc. are identifiable. Having discrete flows of products implies the existence of *operations* on these products. The definition of the set of logistic operations depends on the scope and the level of detail we want to consider. For this purpose, elementary steps are aggregated into operations. Consider, for example, an assembly process. Elementary steps in such a process are: 'fetch tools', 'set-up', 'load part1', 'load part2', 'move robot arm', etc. In most cases we want to model such a sequence of steps

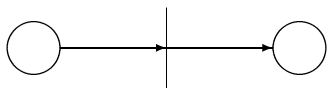
as a single operation. Another example is the production in 'batches', where all elementary steps are per piece. In this case we often consider operations defined per batch instead of per piece. These examples learn that the definition of an operation depends on the desired level of detail.
In a logistic process there are all sorts of operations. When modelling, we often identify the following five attributes of a logistic operation:

1. required goods (materials, goods)
2. required capacity resources
3. processing time of an operation (without waiting times)
4. usage pattern of the capacity resources during the operation
5. produced goods

Generally, these attributes capture the essence of a logistic operation. The firing mechanism of a Petri net allows for the *modelling of an operation in a very elegant and transparent way.* We can think of an operation as a set of *events* and *activities.* Events are represented by transitions. An activity is associated with the firing of a transition or with the presence of a token in a place. An event occurs if all input conditions are met, i.e. each of the input places of a transition contains enough tokens. In this section, we will show that timed coloured Petri nets are well suited to model the five attributes of a logistic operation.

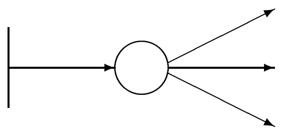
Petri nets also allow for the modelling of *true parallelism* and *synchronization.* By true parallelism we mean that parallelism is clearly differentiated from non-determinism, as opposed to the interleaving of events. To get an impression of the modelling capabilities of Petri nets we show some elementary network structures.

Causality

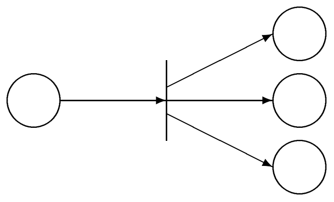


There is a causality relation between the input places and the output places of a transition. No event may be generated 'spontaneously', i.e. without an input event that directly or indirectly caused it.

Divergence There are two kinds of divergence:

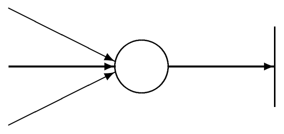


A token produced by a transition is assigned to some other transition in a non-deterministic manner. Only one of the output transitions consumes the produced token. These transitions (events) are said to be in *conflict* with each other. This way it is possible to specify a non-deterministic routing.

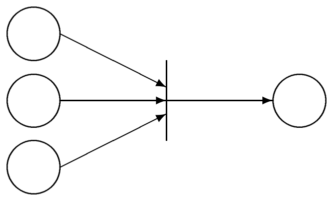


The consumption of a token by some transition results in a number of tokens. This may be interpreted as breaking up an object into a number of (smaller) objects. One can think of a disassembly of a product or an operation having side-effects. An alternative interpretation for this net structure is: one condition implies a number of other conditions. Note that if we use a coloured Petri net model, the number of tokens produced for each output place may depend upon the value(s) of the token(s) consumed. For example, this network structure also matches a 'switch' which sends a token in one out of three possible directions.

Convergence There are also two kinds of convergence:



Several events cause the same result, i.e. there are several ways to meet a condition. This way it is possible to model a converging flow of goods. For example, a number of production units producing products that are stored in the same warehouse.



This is a synchronization primitive comparable to the *join* operation in a computer system. An event occurs if a number of conditions hold. Compare this with the assembly of a number of components into a product.

circuit A *circuit* in a net is a sequence of places and transitions connected to each other such that the sequence starts and ends in the same place. Such a construct is often used to model capacity constraints, reusable materials or a cyclic demand. An example of a capacity constraint is a shared resource, for instance an operator working on a number of machines.

Circuits play an important role in the modelling of logistic systems.

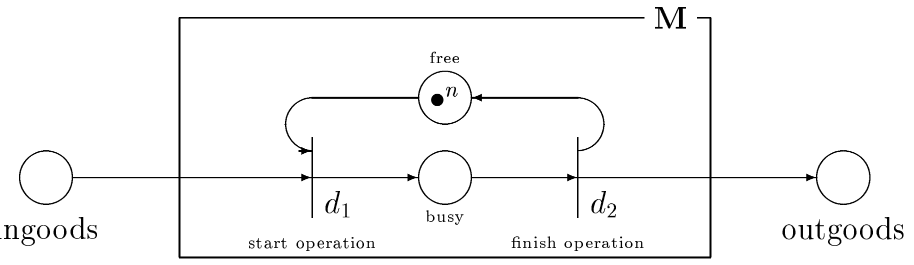


Figure 5.3: A machine having a finite capacity

First of all, they are used to model a *finite capacity*. Consider for example the net shown in figure 5.3. This net represents a machine (or a set of machines) capable of handling n jobs at the same time (n is the number of tokens initially in place $free$). Delay d_1 represents the time a job uses a capacitated resource, delay d_2 is the remaining processing time.

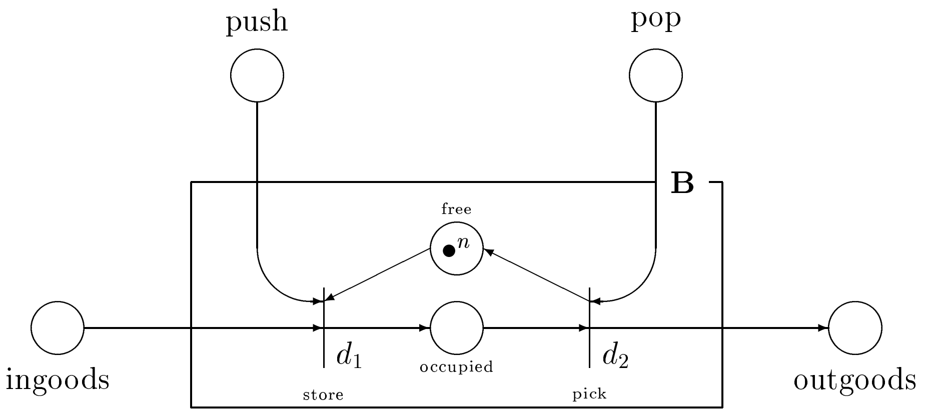


Figure 5.4: A bounded buffer

Another example of a resource with a finite capacity is a buffer of size n, i.e. a bounded buffer (see figure 5.4). The tokens in place *occupied* represent the stored products. The buffer 'releases' a product if there is a token in *pop* and the buffer contains at least one product. To store a product there has to be a token in place *ingoods* and in place *push*, and there has to be enough space in the buffer. We often omit the place *push* to model the property that goods are stored as soon as possible.

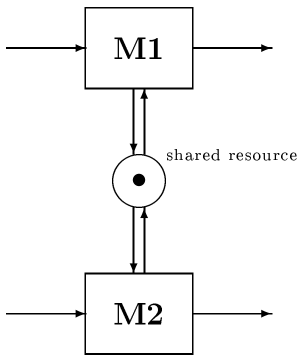


Figure 5.5: A competitive shared resource

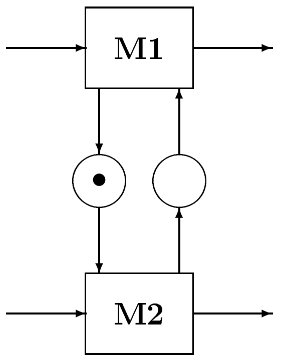


Figure 5.6: A cyclic shared resource

Circuits are also used to model shared resources. Many operations use one or more *shared resources*. Examples of shared resources are: an operator working on a number of machines, loading/unloading facilities and the central computer system. In fact a machine itself can be seen as a shared resource (shared by the different products).

A *competitive shared resource* is a resource shared among a number of processes which may claim the resource at the same time. For example: machine $M1$ and machine $M2$ compete for a resource, see figure 5.5. If both machines want the resource at same time, it is not determined which one wins. It is also possible to model priorities (i.e. one machine comes before the other) or to model a *cyclic shared resource*. Figure 5.6 shows an example of a cyclic shared resource. In this case the resource is used alternately (round-robin). A disadvantage of such a resource is the fact that there can be unnecessary waiting.

An interesting example of a competitive shared resource is a buffer shared by a number of production lines. In this case the storage space inside the buffer is a capacity resource, see figure 5.7. The buffer comprises n units of space (initially there are n tokens in place *free*). A product of type A requires k_1 units of space, a

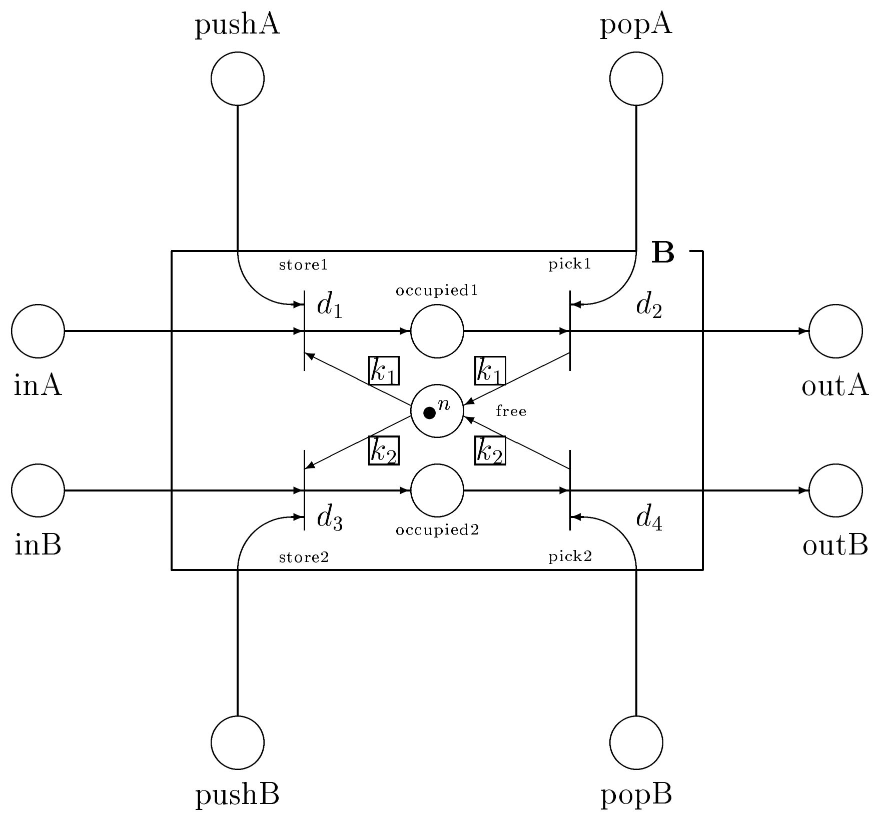


Figure 5.7: A shared buffer

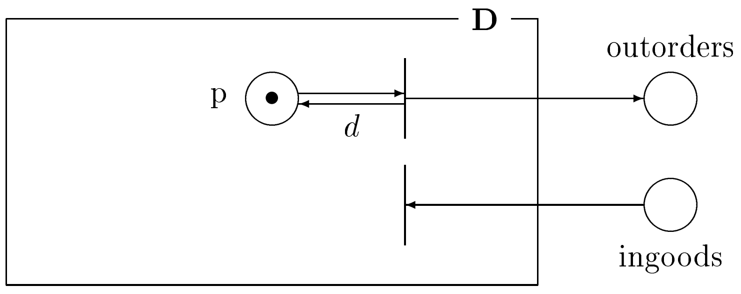


Figure 5.8: A demand process

product of type B requires k_2 units of space ($\boxed{k_1}$ and $\boxed{k_2}$ denote the multiplicity of the corresponding arc). A product of type A can only be stored if there is enough space left, i.e. the number of tokens in *free* is at least k_1.
Finally, we also use circuits to model cyclic processes, for example a demand process. Figure 5.8 shows such a process. Note a token (i.e. an order) is generated every d time units.

The concept of multiple input and output arcs is very handy when modelling the production of batches of products or the assembly of components. Consider, for example, the assembly machine in figure 5.9. This machine uses k_1 products A and k_2 products B, to assemble k_3 products C.

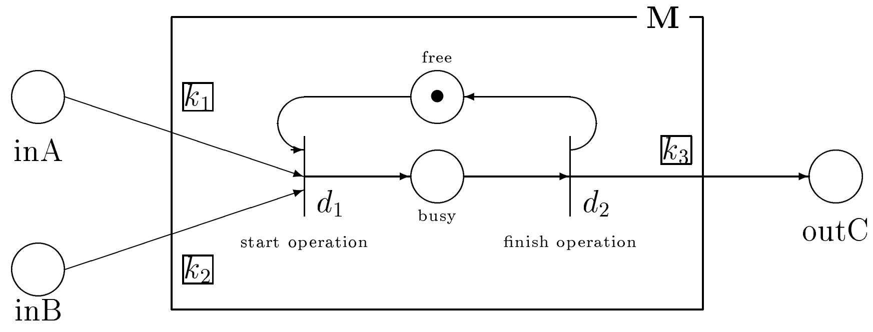


Figure 5.9: An assembly machine

In the examples we assumed that we have a Petri net model with *explicit time*, this allowed us to specify the duration of several operations. Petri nets without time are unfit for the modelling of logistic processes, because time plays a prominent part in logistics. For the sake of simplicity, we used deterministic delays. However, for modelling a real logistic system, we advocate a Petri net model with time in tokens and delays specified by an interval. In chapter 2, we motivated this choice. In our opinion, interval timing is useful when modelling a logistic system, because the precise duration of a logistic operation is often unknown. On the other hand, we want to guarantee a specific logistic performance.
To verify or to estimate logistic performance measures, we need analysis tools. We provide three kinds of analysis: simulation, structural analysis (invariants) and interval analysis (MTSRT, PNRT, ATCFN), see chapters 3 and 4.

To model 'real' logistic systems, we have to use a model with *coloured* tokens, because a token often represents an object having a number of meaningful attributes. If a token represents a product, then it might be useful to model the type of the product, an identification number, its destination, etc. This is the reason we use a coloured (high-level) Petri net model. A coloured Petri net model allows the modeller to make more succinct and manageable descriptions.

When modelling a logistic system with a (timed) coloured Petri net model, we often have to choose between 'putting information in the net structure' and 'putting information in the value of a token'. Putting more information in the net structure results in a larger and more complex Petri net. Putting more information in the value of a token results in more complex operations on the value of a token, and therefore, in a more complex description of the behaviour of some of the transitions in the net. To model a logistic system in terms of a coloured Petri net, we have to balance continuously between the complexity of the net structure and the complexity of the token values.
Consider for example a machine shop with three machines: 1, 2 and 3. The machines are able to process one kind of operation, e.g. drilling. The time required to process a drilling operation is variable. However, for each machine we know an upper and

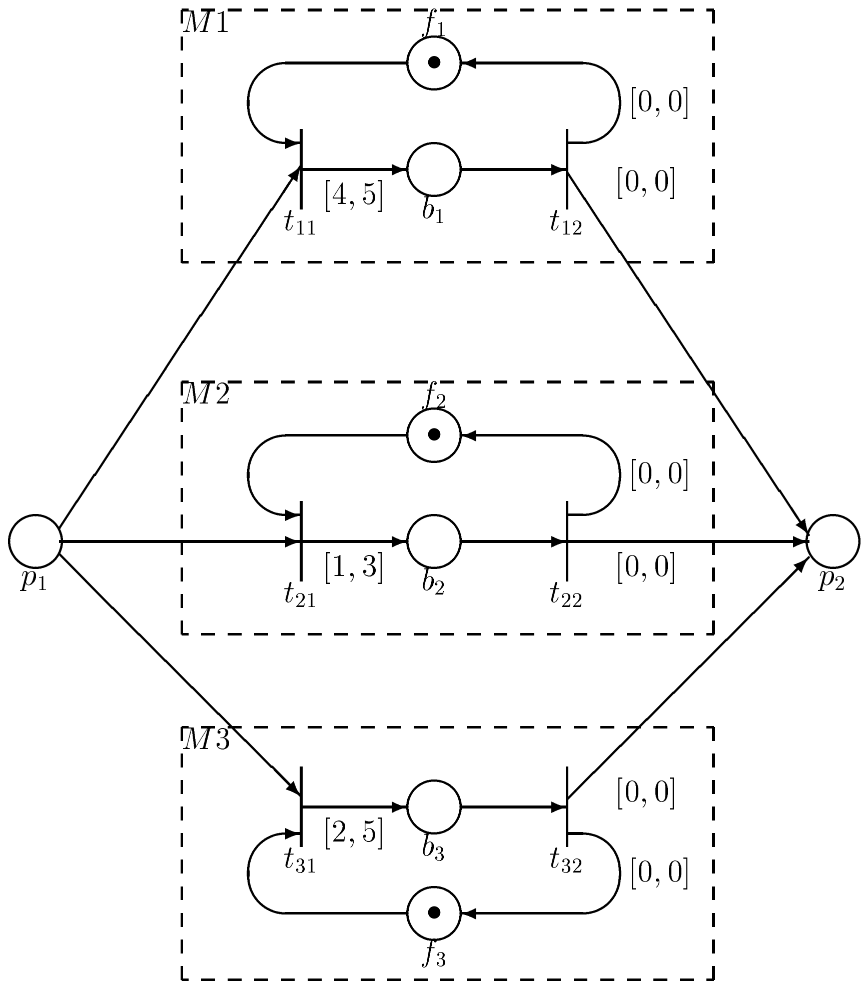


Figure 5.10: Three parallel machines (1)

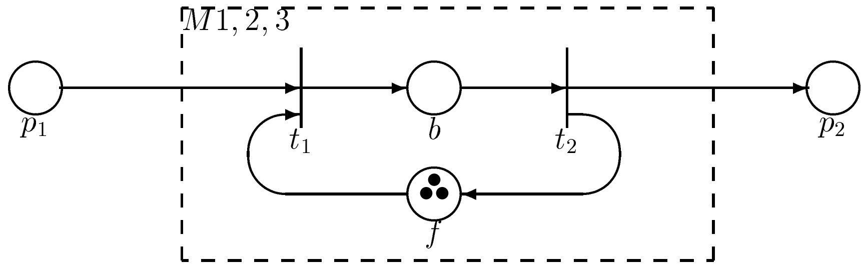


Figure 5.11: Three parallel machines (2)

lower bound for the processing time. We can model this by an ITCPN having the structure shown in figure 5.10. Transition t_{11} (t_{12}) represents the start (end) of an operation performed by machine 1, transition t_{21} (t_{22}) represents the start (end) of an operation performed by machine 2, etc. Initially, there is one token in place f_1 indicating that machine 1 is free, etc. Place p_1 represents a buffer in

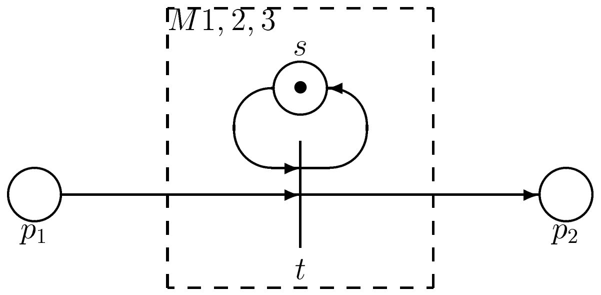


Figure 5.12: Three parallel machines (3)

front of the machines. Assume, it suffices to model machines by ‘colourless’ tokens, i.e. we are not interested in attributes representing aspects like wear, maintenance, disturbances, etc. In this case most of the information about the machines is in the net structure.

Another way to model the jobshop is shown in figure 5.11. Transition t_1 (t_2) represents the start (end) of an operation performed by one of the three machines. Initially, there are three tokens in place f indicating that the three machines are free. To distinguish between the machines these tokens have a value (e.g. 1, 2 and 3).

It is also possible to represent the state of the machine shop by a single token in a place s, see figure 5.12. The value of this token represents information about the three machines.

Finally, it is possible to model the entire system, i.e. the machine shop and its environment, by the net shown in figure 5.13. Note that any ITCPN can be replaced by an equivalent ITCPN which is composed of one place and one transition (like in figure 5.13).

For this example, the nets shown in figure 5.10 and 5.11, seem to be natural. In general, it is difficult to provide guidelines concerning the trade-off between the complexity of the net structure and the complexity of the token values.

Note that this issue is related to the refinement concept. A refinement of a net results in a transfer of information from the token values to the net structure. In section 3.5.2 we discussed the formal relationship between an ITCPN and a refined ITCPN.

Although we have extended our Petri net model with time and colour, modelling a real logistic system in terms of an ITCPN often results in a net which is too large to comprehend. This is the reason a hierarchy construct, called *system*, has been added to ExSpect (see chapter 4). There are some other powerful features which have been added to ExSpect: encapsulation, polymorphism, etc.

Clearly, an approach based on a timed coloured Petri net model and supported by a language (and tools) like ExSpect is suitable for the modelling of large and complex logistic systems. However, we have to compare our approach with more conventional

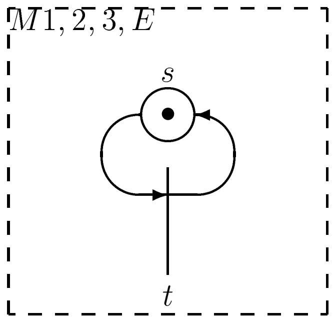


Figure 5.13: Three parallel machines(4)

approaches used to model and/or analyse logistic systems. Alternative formalisms or tools used in the field of logistics are:

data models (ER, relational model)

data flow models (ISAC, SADT, DFD)

analytical models (QN, LP, DP)

simulation languages (SIMULA, SPSS)

specific simulation packages (SIMFACTORY, TAYLOR)

Data models are used to describe complex state spaces, for example a database scheme for an MRP system. Some well-known data models are the entity relationship model (Chen [29]) and the relational model (Ullman [119]). A drawback of these models is the fact that they only describe the static data aspect of a system, i.e. they fail to describe the dynamic structure of a system.

There are several informal frameworks to describe data flow, often using graphical languages. Frequent used frameworks are SADT (Marca and McGowan[79]), ISAC (Lundeberg et al. [78]) and DFD (Ward and Mellor [121]). Most of these frameworks also have methods to describe the data structure. The result of using such an approach is an informal description, that does not allow for quantitative analysis.

Analytical models are mathematical models such as, a queueing model, a linear programming model, a dynamic programming model, etc. Sometimes, we have to simplify the problem statement to be able to use these models. Moreover, modelling a 'real' system in terms of such a mathematical model is often quite difficult and requires expert consultation.

We distinguish between two kinds of simulation languages: (1) general purpose programming languages and (2) block-oriented languages. Examples of general purpose programming languages suitable for simulation are: SIMULA (Dahl and Nygaard

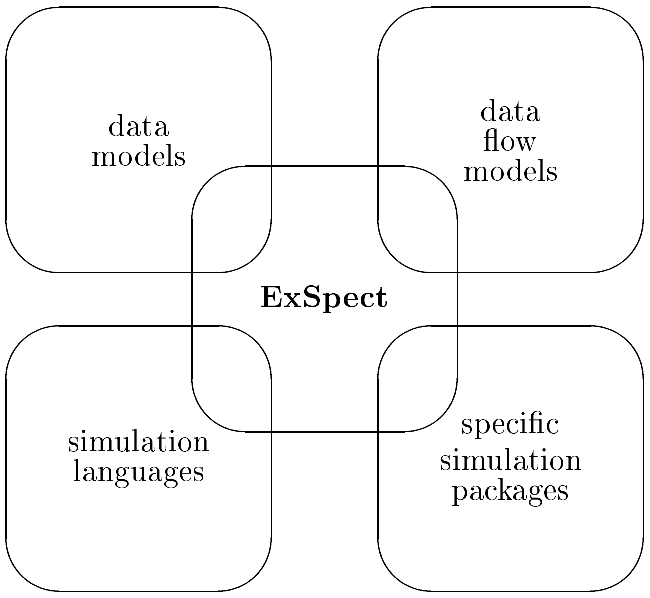


Figure 5.14: ExSpect compared with other tools

[33]) or more conventional languages with libraries of subroutines. Most block oriented languages are based on queueing networks. Examples are SLAM, Q-GERT, SAINT and SIMAN (see Pidd [103]). These languages are flexible and quite fast. However, they are hard to customize, not user friendly and it takes great effort to implement a model. Note that we speak about implementation rather than specification.

Finally, there are specific simulation packages. These packages are application specific. Examples are SIMFACTORY and TAYLOR (Pidd [103]). Most of these packages simulate the internal behaviour of a production unit. These packages are easy to use and have animation facilities. The fact that they are tailored towards a specific application makes them inflexible.

Our claim is that ExSpect combines the advantages of these alternative approaches, as shown in figure 5.14. A token in ExSpect can have an arbitrarily complex type. We are working on the integration of a new data model into our framework (see Van Hee and Verkoulen [58]). Our system concept and the design interface support 'dataflow-like' diagrams. This is very useful in the early modelling phases. Note that we support hierarchical decomposition, comparable to SADT (Marca and McGowan [79]). The module concept allows for the development of domain specific libraries containing generally applicable building blocks. These building blocks tend to be very generic because of polymorphism and parameterization. The specifica-

tion language is very expressive allowing for special purpose constructs, this way it is possible to specify parts of the system not covered by standard building blocks. The tool ExSpect is easy to use because of a mouse oriented interface with 'pop-up' menus. Besides simulation, we also support static type checking and several analysis techniques (e.g. simulation, invariants, MTSRT, etc.). These analysis methods are possible, because we use a formal framework which is based on Petri nets (which have a firm mathematical foundation).
We think it is also possible to use the ExSpect specification as a starting point for analysis using mathematical techniques such as, dynamic programming, linear programming, Markovian analysis, etc. (see Wessels [122] and [123]). To use these analytical models it is often necessary to restrict ourselves to a limited set of specifications. Consider for example a specification composed of the queueing components described in section 4.5. Such a specification may be analysed using analytical techniques developed for queueing networks. Another example is the use of an ExSpect specification of a distribution network, as input for a linear programming model. In this case, the ExSpect specification is used as a 'blueprint' (i.e. a detailed description) of the logistic system. Suitable projections of such a blueprint may allow for analysis using mathematical models. Clearly, the integration of these analytical models into our framework requires a considerable amount of research.

Additional advantages of ExSpect are the open architecture, the software and the possibility to connect several runtime interfaces (running on different machines) to an interpreter, all interacting with the same simulation (ideal for training purposes). See chapter 4 for more information about the features of ExSpect.

Considering the requirements for a specification language for (discrete) logistics, it may be concluded that ExSpect is a sensible choice for the modelling and analysis of logistic systems. However, other Petri net based tools (e.g. CPN [71]) or approaches based on process algebra (see Biemans and Blonk [20] and Mauw [85]) are worth considering. See chapter 1 for a discussion on this subject.

5.4 Structuring logistic systems

In the area of logistics many books are available, nearly all of which deal with the control and design of production, inventory and transport systems. These books reflect the fact that research in the field of logistics developed along two separate lines.
The first line concentrates on solving mathematical problems related to logistics. Investigations in this area are part of a discipline called *operations research*. The models used in this discipline are elegant and allow for powerful methods of analysis. However, it is often difficult to model a real system in terms of such an analytical model. Therefore, the problem statement is often simplified to allow for analytical solutions. Consider for example the application of queueing networks to scheduling problems and the application of linear programming to transport planning. Al-

though these analysis methods help us gain insight in the problem, they can only be applied in rather specific situations or require expert consultation. Moreover, some of the results reported in this area describe techniques for problems that do not even exist.

The second line of research concentrates on practical logistic problems. The results are often qualitative and informal. The approaches used in this area are mainly discipline oriented, i.e. they focus on a specific aspect of logistics. Examples are the research on customer service, storage equipment, communication facilities (EDI), personnel requirements, etc.

Both of these lines did not lead to a complete and comprehensive model of logistics. Recent literature in the field of production control stresses the need for a systematic approach to production planning and control (Bertrand, Wortmann and Wijngaard [18], Biemans et al. [21], [19]). In [19], Biemans attempts to structure manufacturing planning and control using a 'reference model', i.e. a representation of an idealized production organization, defining the tasks of the components as well as the interactions between the components. In [18], Bertrand et al. describe a number of general concepts for the design of production control systems.

In our opinion, there is also a need for a systematic approach to logistics. The main reason to structure logistics is the growing complexity of the control problems in logistics. This complexity is partly caused by the total cost concept, described in this chapter, which forces us to consider the entire logistic chain. Another reason for the increased complexity is the progress in information technology allowing for more sophisticated management systems.

In this chapter we structure the field of logistics by making a first step towards a comprehensive 'reference model' for logistics (see also [4] and [5]). To realize this, we use a systematic approach based on concepts from systems analysis (see section 4.2.4). Similar approaches have been developed for other application domains, e.g. in [35], De Leeuw uses a systems approach to structure organization theory.

Our approach is intentionally *abstract*. Therefore, we focus on the main logistic functions (e.g. transport, demand, supply, production and stock holding) and ignore aspects like administration, safety, personnel, etc. Moreover, sometimes we abstract from the physical reality, i.e. we are not interested in the actual layout of a logistic system, mechanical aspects, communication protocols, etc.

To structure logistics, we identify and specify typical flows and activities in the context of logistics. This results in a taxonomy of the logistic flows and a formal definition of a logistic system. This formal definition is a first step towards a comprehensive reference model for logistics. The term 'reference model' was introduced by Biemans in [19] and [21]. A reference model describes a complex system as a configuration of interacting subsystems (components) that each execute a specific task. Compared to the reference model for manufacturing planning and control, described in Biemans [19], our approach is more formal (and abstract) and addresses another application domain.

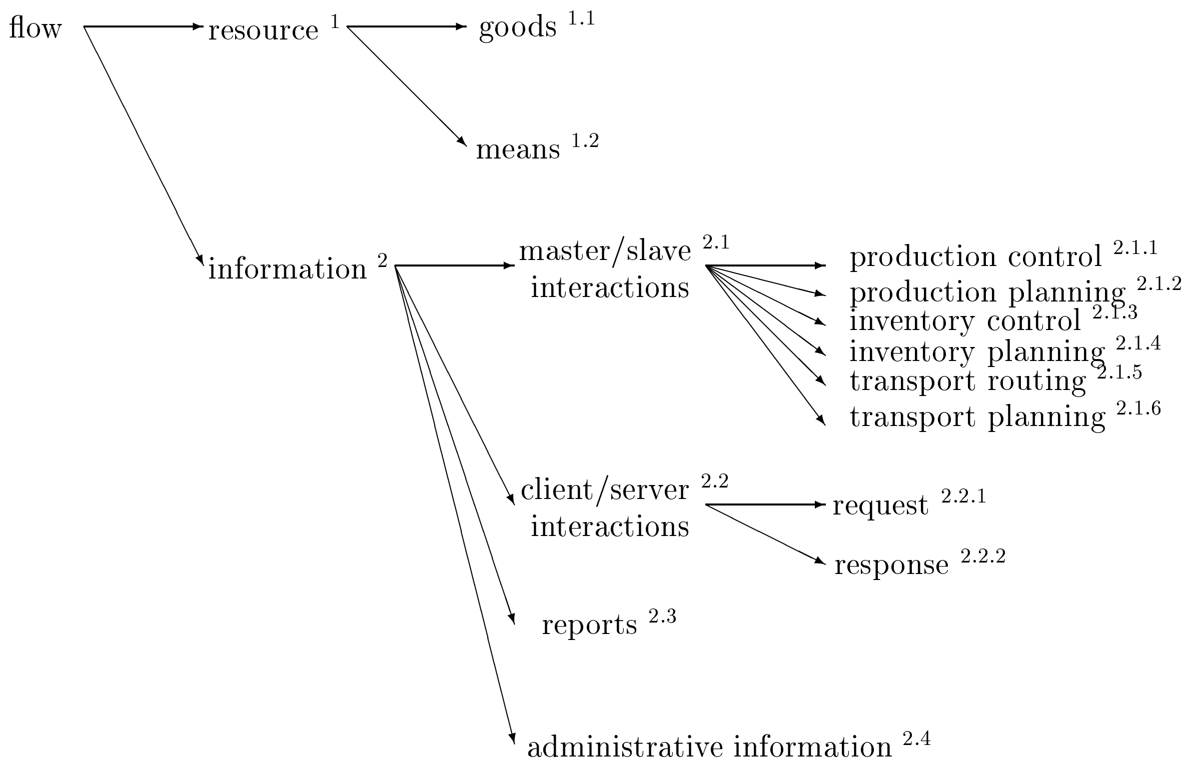


Figure 5.15: A taxonomy of the flows inside a logistic system

Based on our definition of a logistic system, we have developed a logistic library, that will be described in section 5.5. The predefined components in this library are formal specifications of the logistic subsystems identified in this section.

To structure the field of logistics we start with a taxonomy of the flows inside a logistic system. Figure 5.15 shows our taxonomy, the arrows should be interpreted as 'is subtype of'. For example, the flow of goods is a subtype of the flow of resources.

Resources (1) are the physical or abstract objects in a system. We distinguish between *goods* (1.1) and *means* (1.2). Goods are the materials, components and products flowing through the logistic chain. In general these goods are physical objects. Examples of non-physical goods are bank accounts or reservations, we call these objects abstract objects. The resources needed to create, maintain or distribute both kinds of goods are called means, e.g. machines, tools, trucks, manpower, etc. Means are employed, but not consumed like materials. Sometimes we use the term *capacity resources* to refer to these means. It is hard to draw a strict dividing line between goods and means, think for example of a tool in a machine that wears off significantly when it is used. In general, means are *active* and goods are *passive* resources.

We use the term *information* (2) for all other kinds of interaction. Information can be characterized by: 'all the messages needed to get the right quantity of goods at the right time at the right place'. Information itself is not an object to pursue. In most cases information is kept to a minimum. We divide the class of information flows into four subclasses: *master/slave interactions* (2.1), *client/server interactions* (2.2), *reports* (2.3) and *administrative information* (2.4).

Master/slave interactions are the messages exchanged between a control system (master) and a subordinate system (slave). The master sends commands to the slave and the slave sends some status information to the master. Essential is the fact that their relationship is not based on equality. Examples of such interactions are: *(real-time) production control* (2.1.1), *production planning* (2.1.2), *inventory control* (2.1.3), *inventory planning* (2.1.4), *transport routing* (2.1.5) and *transport planning* (2.1.6). For the moment this classification is self-explanatory. Although our classification of master/slave interactions is not exhaustive, we think it covers most control interactions encountered in logistics. We will return to this subject in section 5.5.

Client/server interactions are based on the equality of both parties involved. An alternative term for client/server interactions is coordination. Coordination is based on requests and responses instead of commands and status information. The client sends a *request* (2.2.1) to a server. Typical requests are: the ordering of goods and services, inquiries about the charges and the reservation of capacity resources. Note that placing an order with a supplier is a request rather than a command. A request is always followed by a *response* (2.2.2) from the server to the client. There are two kinds of requests and responses: with and without a 'commit'. A request without a commit means that the client only inquires about some service or goods. Otherwise (with commit), the request is satisfied by the server if possible. In this case there is response with a commit indicating that the server will deliver the requested service or goods. In all other cases there is a response without a commit. Note that this classification conforms with the ideas emerging from the field of Electronic Data Interchange (EDI).

Finally, we have the flows of reports and administrative information. These are the information flows not covered by the flows (2.1) and (2.2). A detailed description of these flows is beyond the scope of this chapter.

We introduce a graphical convention to denote these flows: flows of resources are represented by a double arrow and flows of information are represented by single arrows. To distinguish flows of means from flows of goods, we represent flows of means by dashed double arrows. Client/server interactions are also represented by dashed arrows. All other flows of information are represented by an ordinary arrow. Figure 5.16 shows these graphical notations. This concludes our taxonomy of the flows inside a logistic system. In section 5.5 we will show how to model these flows in terms of ExSpect types.

Figure 5.16 shows the general form of a logistic system. The behaviour of a 'real' logistic system is often too complex to comprehend, therefore we propose a *top down*

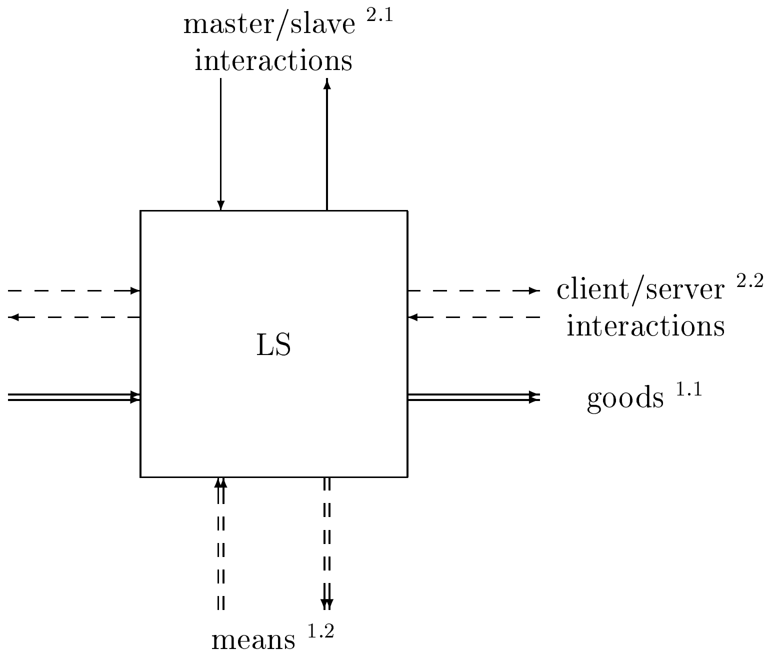
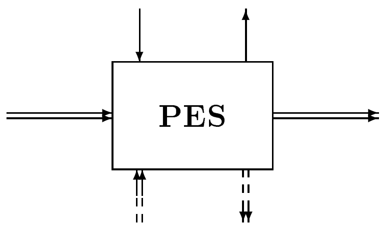


Figure 5.16: A logistic system

approach. This approach deals with the complexity by decomposing the logistic system into subsystems. Each of these subsystems represents a separate logistic process with a distinct task in the context of the overall system. It is possible to repeat this process until the lowest level is reached. At the lowest level there are three kinds of systems:

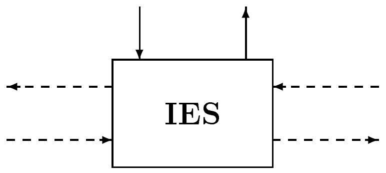
- *physical elementary systems* (PES)
- *information elementary systems* (IES)
- *control systems* (CS)

Physical elementary systems (PES) are systems dealing with resources and are controlled by master/slave interactions. Examples of PES are machines, automated guided vehicles and people doing manual work. Schematically a PES looks as follows:



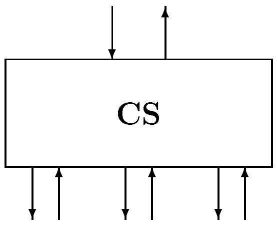
This figure shows a 'typical' PES, e.g. it is also possible to have a PES without master/slave interactions or a PES which does not exchange means with its environment.

Information elementary systems (IES) are systems dealing only with information. An IES is also controlled by master/slave interactions. Schematically:



Examples of IES are demand forecast and order entry systems. An IES is controlled by some higher authority and communicates with other (information) systems via requests and responses (client/server interactions).

Elementary systems (PES and IES) are controlled by a control system (CS). A control system controls subordinate systems via master/slave interactions and is controlled by master/slave interactions. Examples of CS are: real-time controllers, MRP-modules and managers. In general an incoming command is translated into a number of commands for the subordinate systems. Schematically:



Now we can give a recursive *definition of a logistic system* (LS): a logistic system is an elementary system (PES or IES) or a set of logistic systems controlled by a control system (CS). Figure 5.17 shows an example of a logistic system. Our definition of a logistic system (LS) is summarized in figure 5.18. Physical elementary systems and information elementary systems are logistic systems. A group of logistic systems is a logistic system. One or more logistic systems controlled by some control system is also a logistic system. In [50], Van Hee and Somers use a similar recursive definition of a production system.

Our top down approach produces a *hierarchy* of systems. A logistic system, which is too complex to comprehend, is decomposed into a number of logistic subsystems. This decomposition process is repeated until the logistic subsystems are considered elementary.

The definition of a logistic system summarized in figure 5.18 and the taxonomy

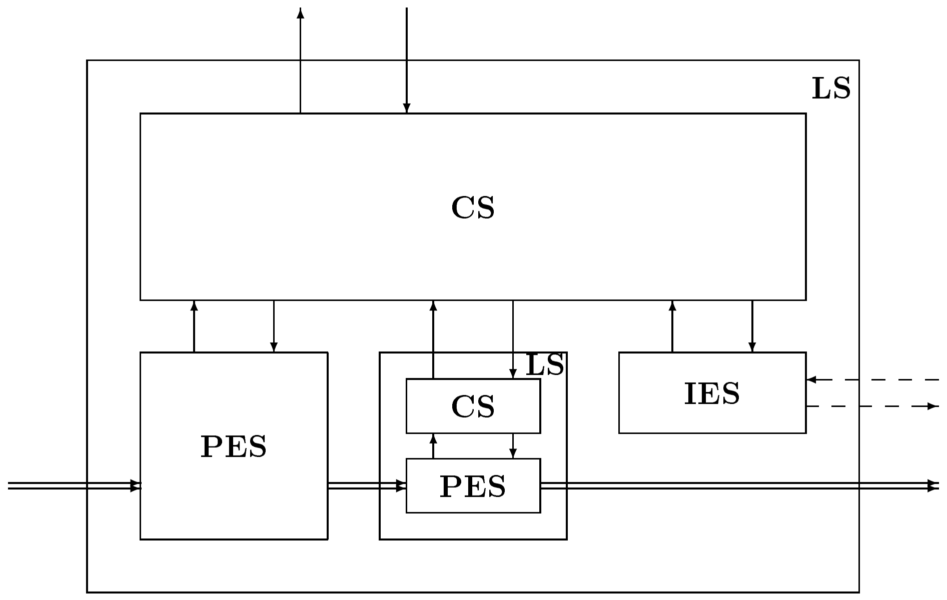


Figure 5.17: A logistic system

`PES =`	*physical elementary system*
`IES =`	*information elementary system*
`CS =`	*control system*
`LS =`	`PES | IES | LS-list | CS,LS-list`

Figure 5.18: A recursive definition of a logistic system

shown in figure 5.15 constitute a basis for a *reference model for logistics.* The recursive definition of a logistic system tells us how to decompose a logistic system into meaningful subsystems, i.e. decompositions have to meet the criteria stated in figure 5.18. Based on these criteria, we find relatively independent subsystems executing a specific task and having a typical interaction structure. Expressing the interaction structure in terms of the flows identified in figure 5.15, helps us to find characteristic components. Identifying a limited set of characteristic components, large enough to represent most of the logistic systems encountered in practice, yields a reference model. The development of a comprehensive reference model for logistics requires a lot of research and experience with the modelling of many real logistic systems. We are convinced that this is possible, this is fortified by the existence of an informal reference model for production planning and control presented by Biemans in [19].

Clearly, the development of such a reference model is beyond the scope of the research reported in this monograph. Instead, we give a short informal description of the typical logistic activities and control structures encountered in practice. In section 5.5 we will map the activities and control structures onto components specified with ExSpect. The result is a logistic library. With this this library we hope to attain a '80/20'-situation, i.e. a situation where 80 percent of the components needed are already available in a logistic library and take up only 20 percent of your time. But the 20 percent you have to create yourself take up 80 percent of your time. It is obvious, that it is not possible to attain this situation without a rigorous structuring of logistics. Moreover, the development of a library based on a comprehensive reference model for logistics, would yield a situation where nearly all of the components needed are already available (e.g. a '94/15'-situation [1]).

5.4.1 Typical logistic activities

As stated, we confine ourselves to a short informal description of the typical logistic activities and control structures. We start with a review of the primary logistic activities: (1) demand, (2) supply, (3) transport, (4) transformation and (5) inventory. The logistic library described in section 5.5 contains a component (building block) for each of these activities.

Demand

Demand is the trigger for all logistic activities (although demand may be stimulated by marketing). The demand for end-products is generated by a number of consumers. In our opinion the identity (or definition) of a consumer depends upon the scope of the logistic chain we want to consider. Suppose we have an audio manufacturing firm supplying a number of wholesale dealers. Each wholesale dealer supplies a number of retailers, and finally, each retailer sells audio equipment to its customers. Depending on the scope of the logistic chain we want to consider, we define the wholesale dealers *or* the retailers *or* the customers to be the entities that generate the demand.

The demand (for a specific product) is often instable and subject to trends and seasonal patterns. If there is a frequent ordering of small quantities, we speak about *independent demand*. If there are only a few consumers ordering (relatively) large quantities or there is a strong correlation between the demand for a number of products, we speak about *dependent demand*.

It is often useful to classify the products demanded by the consumers into three classes: A, B and C. Class A contains products having a high demand, class B represents products having an 'average' demand, products in class C are ordered sporadically. In most situations a small percentage of the products account for a large percentage of the total demand, i.e. the products in class A represent the main part of the demand. This classification process is often called *ABC-analysis*.

[1]94 percent of the components needed are already available in a logistic library and take up only 15 percent of your time. But the 6 percent you have to create yourself take up 85 percent of your time.

Supply

The supply process takes care of the input of raw materials and components into the logistic chain. The identity of a supplier also depends upon the scope of the logistic chain we want to consider. The performance of supplier is measured in terms like: lead time, variations in lead time, product quality, capacity and price.

Transport

Transport is a key factor in today's logistics. We distinguish between two kinds of transport: *internal transport* and *external transport*. Internal transport is the transport inside a plant or warehouse, external transport moves goods between plants and warehouses. This distinction is not absolute, for example it is difficult to classify the transport between two production units.
The forklift truck is the most popular transportation aid in internal transport. It is often used in conjunction with pallets. Next in popularity is the conveyor. Conveyor systems are particularly useful for moving items along a fixed route. There are a number of different conveyor types (wheel, roller or belt) to accommodate specific needs. A relatively new way to transport materials inside a building is the Automated Guided Vehicle (AGV).
There are five basic transportation modes for external transport: rail, highway, water, pipeline and air. Each transport mode has its own characteristics. For example, transport via water is slow but cheap for high volumes, transport via highways is more expensive but faster and more flexible. The selection of transport mode depends upon the products to be transported, required speed, locations and costs.
Note that from a modelling point of view the transport mode is not important, only the relevant characteristics matter.

Transformation

A transformation process uses one or more resources to produce one or more (possibly different) resources. One can think of a step in a manufacturing process or the servicing inside a bank or hospital.
The two main characteristics of a transformation process are the speed and capacity. Note that a step in a manufacturing process may result in a converging or diverging flow of goods. For example, an assembly process combines several types of products into one product.
Remember that we distinguish between two kinds of resources: goods and capacity resources (means). If a number of transformation systems share a capacity resource, we speak about a *shared resource*. Examples of shared resources are: manpower, machines, etc.

Inventory

Inventories are needed for a number of reasons:

Production needs Warehousing may be part of the production process because certain products require a period of aging. For example, painted products have to dry and cheese has to age.

Coordination of supply and demand If there is a seasonal demand or production, then supply and demand have to be coordinated. For example, canned fruits have a constant demand and a seasonal production. Therefore, companies producing canned fruits have to stockpile production output in order to meet the demand during the rest of the year. To guarantee a fast delivery, if necessary, end-products are stored close to the customer.

Costs Transportation of large volumes is relatively cheap. Production in large batches reduces the production costs. Therefore, it is sometimes possible to reduce transportation or production costs by trading them for warehousing costs.

Inside a production process there are buffers to allow machines to continue working while another machine undergoes maintenance, tool changes, or repairs. The materials contained by these buffers are referred to as *in-process inventory*.

Inventory ties up capital, uses storage space, deteriorates and sometimes becomes obsolete. Therefore, the main objective of inventory management is to minimize the inventory without disturbing the production or distribution process. To conclude, we mention that there are two alternative interpretations of inventory: 'transport with speed zero' and 'transformation in time'.

5.4.2 Typical control structures

We define *logistics control* as the coordination of the logistic activities to achieve a specific external performance at minimum costs. This coordination is often difficult, because there are conflicting objectives. For example, economic objectives are often in conflict with customer service objectives or flexibility objectives.
One way to avoid sub-optimal control is to centralize the control function. However, it is hard to centralize the control of a complex logistic system without a large investment in information systems. Instead of a centralized approach it is also possible to create self-contained activities. Using self-contained activities simplifies the control function. A hierarchical approach combines the advantages of a centralized control and self-contained units. The application of a hierarchical approach to production control is advocated by a number of authors (see Meal [87] and Bertrand at al. [18]).
We distinguish four typical control structures in logistics: (1) local control, (2) push control, (3) pull control and (4) integral control. We will show that these control structures fit in the framework described in this section. At the same time, we illustrate that it is possible to use this framework to express recent developments in logistics (e.g. JIT, MRP, DRP, BSC, Kanban).

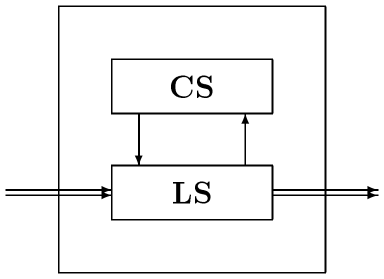


Figure 5.19: Local control

Local control

A classical approach towards logistics control is the creation of a completely self-contained units. Such a unit is only driven by the arrival of goods. This situation is depicted in figure 5.19. Many intermediate steps in a manufacturing process have 'local control'. An example of such a process is an assembly conveyor.

Push control

A set of successive logistic activities is controlled by 'push control', if the first activity is controlled by a master plan and all intermediate activities have a local control (see figure 5.20). This master plan is based on demand forecasts and initial inventories. This kind of control is easy to realize but it suffers from a number serious drawbacks: low flexibility and large inventories or a poor delivery performance. These drawbacks are the result of the absence of real-time feedback from the demand. Examples of 'pure' push systems are found in the field of continuous production, engineer-to-order and production-to-stock (driven by forecasts) systems.

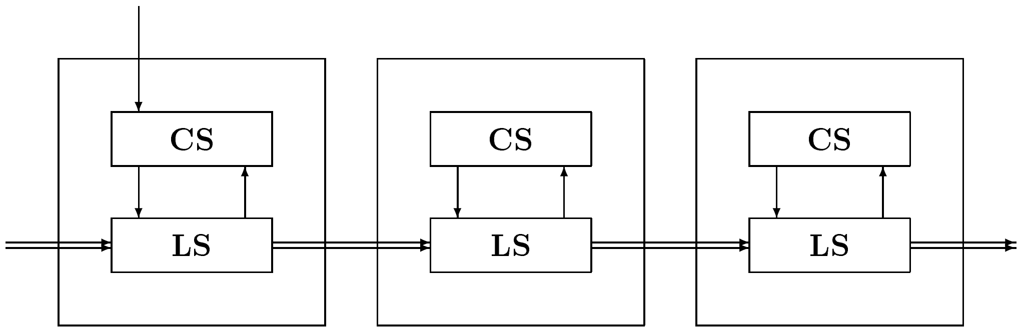


Figure 5.20: Push control

Pull control

A 'pull system' is a system where all activities are triggered by demand (see figure 5.21). Pull controlled systems are demand driven: a production or supply action is issued at the moment a product is requested or inventory is below a given value. The classical inventory management systems, often referred to as *Statistical*

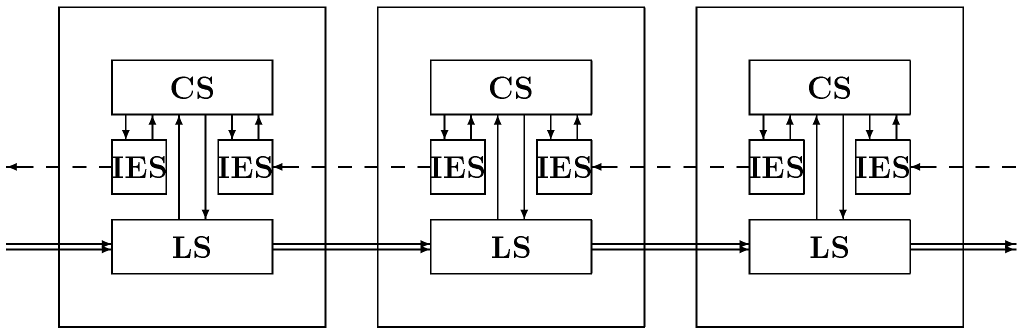


Figure 5.21: Pull control

Inventory Control (SIC) systems, are examples of pull systems. The objective of these systems is to replenish stocks at the 'right moment' in the 'right quantity'. There are basically two ways to determine when to order: (1) order at the moment the stock falls below a (fixed) minimum level called the order point (B) or (2) check the inventory periodically, i.e. if stock is below a certain level (s), a replenishment order is issued. There are two ways to determine the quantity: (1) a fixed quantity (Q) or (2) a quantity depending on the current operating stock (S). An example of an (s,S) inventory management system is a system checking the stock at the end of every month and the order quantity is the difference between a predefined maximum level and the current operating stock.

We already mentioned a number of reasons for the existence of inventory. Some of these reasons are in conflict with the *Just-In-Time* (JIT) philosophy. The goal of the JIT approach is to reduce inventories ('zero inventory') and waste ('total quality control') by obtaining or producing, just what is needed, just when it is needed. Removing excess inventory and inspection forces problems to surface. The JIT approach tries to solve these problems continuously. The rise of JIT is closely related to the success of the Japanese industry and the development of the *Kanban* production system. The Kanban system, developed at the Toyota Motor Company, uses a pull control. This pull control is implemented using two kinds of cards (*kanbans*): withdrawal (or transport) kanbans and production kanbans. The withdrawal kanban shows the quantity of products that the subsequent process should withdraw from the preceding one. The production kanban shows the quantity that the preceding process should obtain or produce.

Consider, for example, we have a process A followed by a process B as shown in figure 5.22. Products flow from process A to process B via a store I. Initially there are a number of free withdrawal kanbans in B and a number of free production kanbans in A. Process A produces the products associated with the production kanbans, attaches the kanbans to these products and stores them in the storage location. Process B takes a free withdrawal kanban to the storage location (I), withdraws the required number of products, detaches the production kanban and attaches the withdrawal kanban. The withdrawal kanban becomes free if the corresponding products have been used by process B. Note that this way the in-process inventory is limited

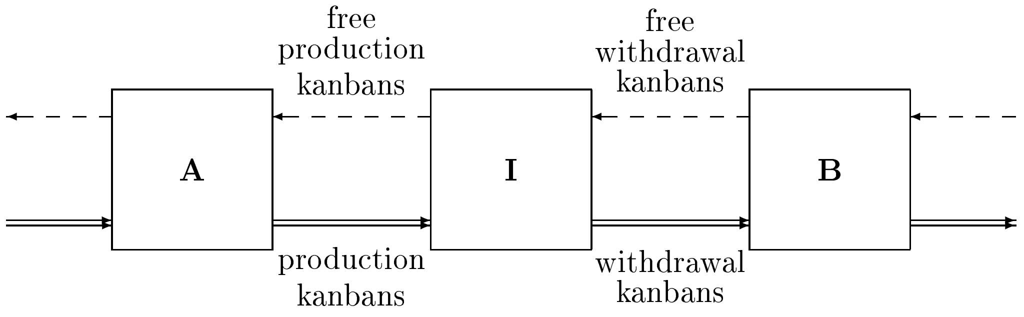


Figure 5.22: A kanban system

by the number of kanbans. The JIT philosophy aims at a continuous reduction of the number of kanbans by improving the production process. There are a number of alternative kanban systems, for example, a kanban system with only one type of cards (kanbans).

The Kanban system of in-process inventory control works particularly well in situations with small batches and a continuous demand.
Both the JIT approach and the introduction of the Kanban system require set-up time reductions, improved quality control and employee involvement and flexibility. Note that information processing hardly plays a role.

Integral control

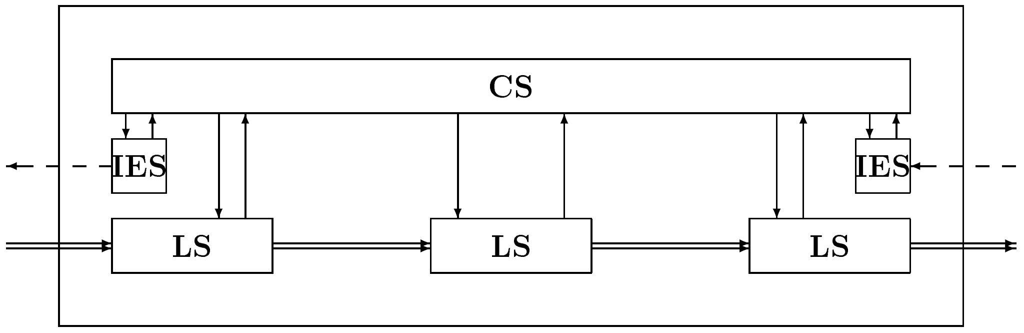


Figure 5.23: Integral control

The increasing availability of computing power stimulated a more centralized approach towards logistics control. Such a centralized control is used to integrate the control of a number of logistic processes. Examples of integral control are: MRP (Materials Requirements Planning), MRP-II (Manufacturing Resources Planning), OPT (Optimized Production Technology), BSC (Base Stock Control), DRP (Distribution Requirements Planning) and DRP-II (Distribution Resources Planning). Other examples of integral control are found in the field of computer aided manufacturing (CAM) and flexible manufacturing.

Perhaps the most widespread form of integral control is *Materials Requirements*

Planning (MRP). MRP produces a production schedule given: (1) the Bill-Of-Materials (BOM), (2) current inventory, (3) lead times and expected demand for final products (Master Production Schedule). The Bill-Of-Materials (or goes-into graph) is a graph specifying the required products and materials needed in each production step. This combined with the production lead times allows MRP to 'explode' the requirements into a production and purchase schedule. The MRP mechanism has been extended in several ways: safety stocks, minimal batch sizes, failure rates, etc. A serious drawback of MRP is that it does not take capacity constraints into account. This is the main reason for the development of *Manufacturing Resources Planning* (MRP-II). MRP-II checks whether it is possible to meet the Master Production Schedule (MPS) using a rough-cut capacity planning. If there exist serious capacity bottlenecks, the MPS is repeatedly adapted until the MPS is feasible. Using the MPS, it is possible to anticipate a future trend and to balance the load on bottleneck machines. Note that the MPS is no longer a direct translation of external demand. Figure 5.24 shows the two level control hierarchy of MRP-II.

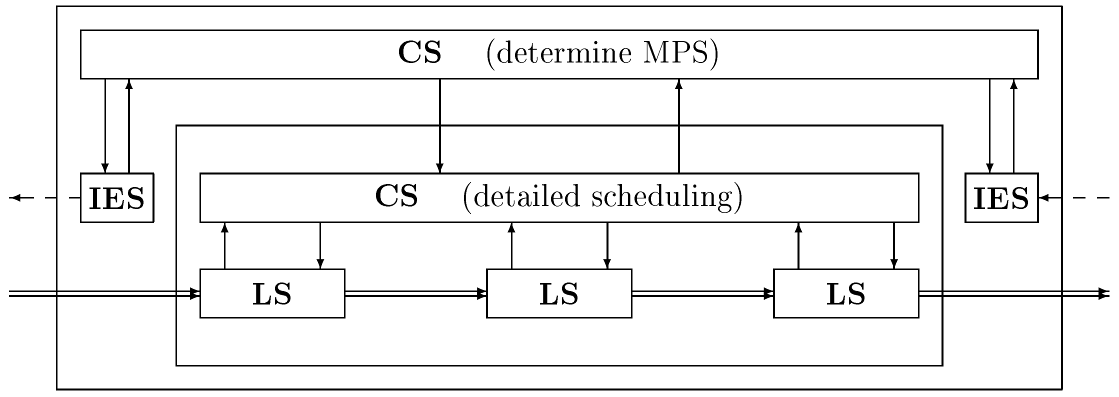


Figure 5.24: MRP-II

OPT is an approach based on a scheduling system. The main characteristic of this approach is the emphasis on the efficient use of bottlenecks.

Distribution Requirements Planning (DRP) is a technique to determine when, where and how to replenish in a distribution network. A typical distribution network controlled by DRP consists of factories, a central warehouse, regional warehouses and retailers. DRP uses: (1) current inventories at each location, (2) transport and handling times and (3) expected demand at each location, to determine a replenishment plan. This plan tells how and when products should be moved among the various locations in the distribution network. DRP applies the MRP principles and techniques to distribution instead of production. Distribution Requirements Planning is often used in combination with MRP, in this case DRP generates the MPS. It is also possible to extend DRP to *Distribution Resources Planning* (DRP-II). DRP-II checks whether the distribution system can handle the generated plan. If not, the plan is revised until all capacity constraints are satisfied.

Another way to manage inventories is *Base Stock Control* (BSC). BSC is closely related to Statistical Inventory Control. In a BSC system all warehouses are aware of the actual demand and the stock levels of the warehouses 'downstream'. In case of a central warehouse supplying a number of regional warehouses, the central warehouse knows the actual demand and the stock levels of the regional warehouses. BSC reduces the total inventory in the distribution system and is not subject to shock waves of unexpected demand.

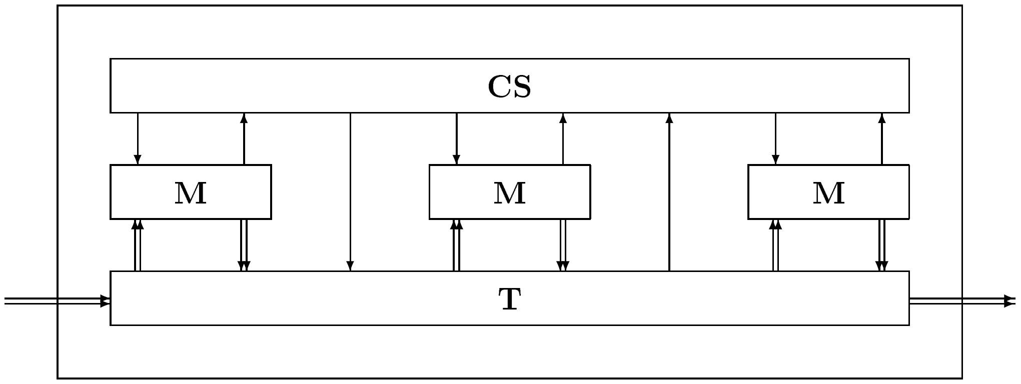


Figure 5.25: A Flexible Manufacturing System (FMS)

Inside a production unit or warehouse there are all sorts of integral control. Think for example of the scheduling of a jobshop or a Flexible Manufacturing System (FMS). An FMS is formed of a set of flexible machines, an automatic transport system and a sophisticated control system to decide at each instant what has to be done on which machine. Figure 5.25 shows a schematic view of an FMS, the subsystem named T represents the transport system, the subsystems named M represent the machine cells. Note that a product follows a route which is not restricted by the physical layout of the shop floor.

This completes our brief review of existing control techniques in logistics. For more information on production control, the reader is referred to Biemans [19], [21], Bertrand et al. [18], Fogarty and Hoffmann [42]. Although many of the concepts for production control apply to logistics control, the management of inventories and transport requires some more attention. Therefore, we focus on the location of inventory and typical distribution structures in a logistic system.

A way to characterize a logistic system is to identify the location of inventories. In the field of production logistics we see three typical structures: (1) make-to-order, (2) assemble-to-order and (3) make-to-stock. In a make-to-order situation, the production of a product starts at the moment an actual demand occurs. In this case there is no inventory of finished and semi-finished products. In a make-to-stock situation, demand for finished products is sufficiently large to allow for production in

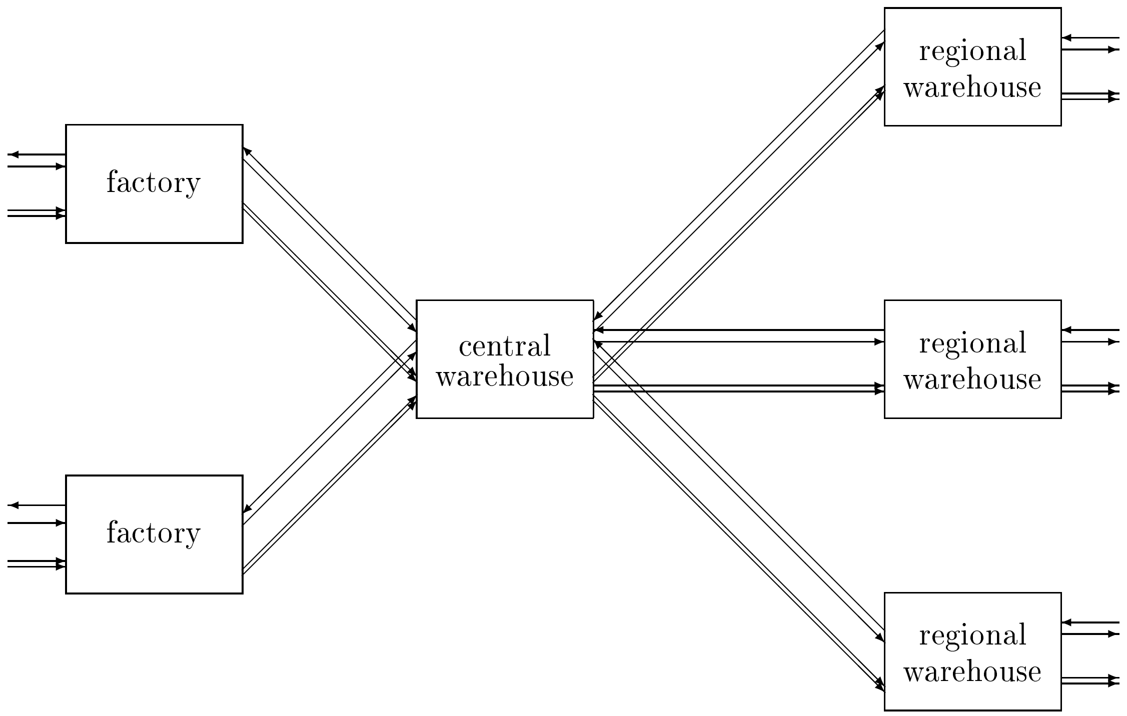


Figure 5.26: A typical distribution structure

advance of actual demand. Assemble-to-order means that subassemblies are manufactured in advance and the assembly of end-products starts on the basis of actual demand.
These three situations can be defined in terms of the location of the so-called *decoupling points*. A decoupling point holds inventory to decouple demand from production or supply. This inventory is replenished by a planned production or delivery. Products are withdrawn from this inventory on the basis of actual demand. For example, a make-to-stock situation is characterized by a customer order decoupling point, which is near the customer and holds end-products.

In the field of physical distribution there are three typical structures: (1) direct delivery, (2) a central warehouse and (3) regional warehouses. In the direct delivery distribution structure, products are supplied directly to the customers without holding inventories in separate warehouses. Sometimes a number of factories supply a central warehouse. This central warehouse supplies the customers. To provide a specified level of customer service these distribution structures utilize high-speed transport. Another possibility is to create a number of regional warehouses close to the customers. Figure 5.26 shows a distribution structure with one central warehouse and a number of regional warehouses. The flow of goods is represented by double arrows, the flow of information is represented by single arrows. Note, that

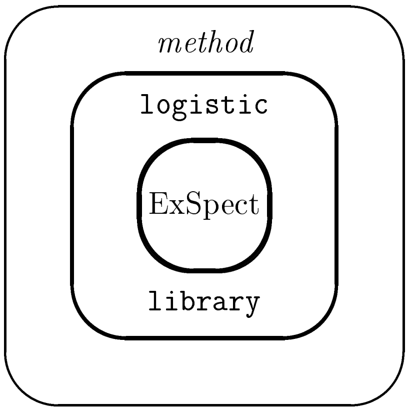


Figure 5.27: The proposed logistic framework

this figure does not specify what kind of information (master/slave interactions, client/server interactions, reports or administrative information) is exchanged between the various locations.

For more information on logistics control we refer to Bowersox [24], Fogarty and Hoffmann [42].

5.5 A logistic library

Based on the approach described in the previous section, we have developed a small logistic library. This library contains a number of generally applicable logistic components.

In section 4.4 we discussed the purpose of such a domain specific library. The two main reasons to develop a logistic library are:

- a logistic library facilitates and speeds up the modelling process
- a logistic library can be used to capture and distribute logistic knowledge

However, for the logistic application domain, a logistic library is not sufficient. To support the use of the library, we have to supply a method. This method tells you, how to use the logistic components (see figure 5.27). We have developed a rather simple method based on the approach described in the previous section. This method is outlined in section 5.6.
Basically, our logistic library consists of two parts: (1) a number of type definitions to model the flows of resources and information, and (2) a number of generic system definitions to model typical logistic activities. The type definitions are based on the taxonomy shown in figure 5.15. Because of the graphical nature of ExSpect, we use the term 'component' or 'building block' to denote a predefined system definition.

We have used the definition of a logistic system shown in figure 5.18, to identify useful components. This implies that a component is: (1) a physical elementary system, (2) an information elementary system, (3) a control system, (4) a system composed of a set of relatively independent logistic components, or (5) a system composed of a set of logistic components controlled by a control system.

The usefulness of the logistic library highly depends upon the utility of the individual components. A building block (component) is considered to be useful if it is:

- easy to use
- powerful
- flexible
- robust

A component is easy to use, if it is easy to understand its semantics and there is a straightforward relation with the world we want to model. This is only possible if the component represents a typical logistic activity with a relatively independent task. The modelling power of a library depends on: (1) the expressive power of the building blocks (is it possible to model something?) and (2) the average size of a model in terms of the building blocks. Note that it is possible to have building blocks allowing for the modelling of a large class of systems, but in a roundabout way. Compare this to programming in assembler, it is possible to program anything, but it takes a lot of effort. The flexibility of a component also depends on two aspects: (1) is it easy to adapt the component and (2) are the important characteristics of a component parameterized. Parameterized building blocks are useful, because they can be tailored for a specific situation, i.e. parameterization is used to make a component generally applicable so that it can be used in a wide variety of applications. Finally, a building block has to be robust in the sense that it can handle various inputs, i.e. the number of assumptions about the environment of the component has to be as small as possible.

Besides the usefulness of the individual components, the conceptual integrity of the library is important. This means that it has to be possible to compose components into a system having a ‘natural’ structure.

The logistic library described in the rest of this chapter tries to maximize the five objectives: easy to use, powerful, flexible, robust and conceptual integrity. Note that some of these objectives may be contradictory. Our goal is not to present an exhaustive list of logistic components covering all situations encountered in logistics, but to show that it is possible to create a comprehensive set of generic logistic building blocks. Our aim is to capture logistic knowledge in this library and to validate the ‘80/20-situation’ described in the previous section.

The library we propose is *hierarchical*, i.e. some of the building blocks are composed

```
type id from num;
type location from str;
type prod from str;
type operation from str;
type capacity from real;
type timewindow from real >< real;
type commit from bool;
type conditions from real;
type age from real;
type material from prod -> real;
type task from operation >< capacity;
type route from (num -> (location >< $task)) >< num;
```

Table 5.1: Some basic type definitions

of other building blocks. ExSpect supports the user of this library in making his own building blocks from already existing ones. This way the user is enabled to make complex hierarchical models with a lot of levels. Therefore, we provide some guidelines: (1) the number of levels in the hierarchy (visible to the user) should be smaller than 6, (2) the number of different building blocks at the same level (in a subsystem) should be smaller than 10. In other words: avoid a shallow or extremely deep hierarchy. Note that these figures are only guidelines, they depend on the system to be modelled.

5.5.1 The type definitions

In section 5.4 we presented a taxonomy of the flows inside a logistic system. We will use this to classify the type definitions used by the logistic building blocks. A list of basic type definitions is given in table 5.1.
The type `material` is a mapping from products (`prod`) to reals representing the quantity of each product. The type `timewindow` is used to denote an interval of time. Another interesting type is the type `route`. A route is a list of pairs and a pointer pointing to a pair in the list. Each pair is formed of a location and a set of tasks. The pointer is used to identify the current location and the tasks to be executed at this location. Note that the list is implemented as a mapping from `num` to `location >< $task`. Table 5.2 shows a value of type `route`.
We have defined some standard functions for this type:

route				
num	location	$ task		num
		operation	capacity	
1	'EindhovenDC'			2
2	'ParisPU8'	'drillingFA8'	2.55	
		'grindingDR7'	1.08	
		'grindingRT6'	1.29	
3	'LyonPU9'	'paintHG9'	4.93	
		'polishIR7'	0.08	
4	'MadridDC'			

Table 5.2: A value of type `route`

```
– 1.1
 type goods from id >< route >< material;
– 1.2
 type means from id >< (operation -> capacity) >< age;
– 2.1.1
 type realtimeprodcommand from material >< means >< task >< material;
 type realtimeprodsignal from material >< $means;
– 2.1.2
 type aggprodcommand from prod -> ((timewindow -> real) >< conditions);
 type aggprodsignal from (prod >< timewindow) -> real;
– 2.1.3
 type delivercommand from goods;
 type receivesignal from goods;
 type stocklevel from material;
 type acceptedorder from goods >< timewindow;
 type replenishcommand from (prod >< timewindow) -> real;
 type replenishsignal from material;
 type ordervolume from ((prod >< timewindow) -> real) >< (material);
 type orderlimit from prod ->((timewindow -> real) >< conditions);
– 2.1.4
 type replenishmentstrategy from prod -> (str >< real >< real >< real);
 type inventorylevels from prod -> (real >< real >< real);
– 2.1.5
 type routecommand from (num -> (location >< $goods >< $goods)) >< means;
 type routesignal from means >< location;
 type availabletranscap from timewindow -> (operation ->
                                            (capacity >< conditions));
 type acceptedtransorder from goods;
– 2.1.6
 type transportstrategy from str >< real >< real >< real;
 type transportperformance from real >< real >< real;
– 2.2
 type request from id >< route >< material >< timewindow ><
                  conditions >< commit;
 type response from id >< route >< material >< timewindow ><
                   conditions >< commit;
– 2.3
 type report from str;
– 2.4
 type admin from str;
– internal types
 type billofmaterial from prod -> (material >< task);
```

Table 5.3: Some logistic type definitions

```
export current[ x : route ] :=
   pi1(x).pi2(x)
 : location >< $task;

export atend[ x : route ] :=
   all([i : dom(pi1(x)) | i <= pi2(x) ])
 : bool;

export atstart[ x : route ] :=
   all([i : dom(pi1(x)) | i >= pi2(x) ])
 : bool;

export next[ x : route ] :=
   pi1(x).min(set([i : dom(pi1(x)) | i > pi2(x)]))
 : location >< $task;

export prev[ x : route ] :=
   pi1(x).max(set([i : dom(pi1(x)) | i < pi2(x)]))
 : location >< $task;
```

All other types definition in table 5.1 are self-explanatory.

Table 5.3 shows some other type definitions, each corresponding to a specific kind of flow in a logistic system. The flow of goods is represented by the type `goods`. Goods flowing through the network have an identification, some routing information and some materials associated with it. Examples of objects of type `goods` are: a truck load, a pallet, a parcel or a single product. Table 5.4 shows a value of type `goods` representing a set of parts, needed to produce a car with identification 897654. Note that currently the parts are located in Paris, where they have to be assembled.

Objects of type `means` have an identification, an age and a capacity for each kind of operation the object can perform. This type is used to specify capacity resources, such as machines, trucks, etc.

Client/server interactions are represented by objects of the type `request` and `response`. A request has an identification, a route, a contents (material), a time window, a condition and a commit field. The usual interpretation of a request is: 'can you deliver me some materials within a time window, given some conditions'. If the commit field is 'true', then the request is automatically satisfied if possible. The `conditions` field is used to specify the requested conditions, for example maximal price or minimal quality. In all cases a request is followed by a response having the same identification.

The other types (mainly master/slave interactions) will be discussed when we describe the corresponding building blocks. Note that we chose 'the easy way out' to model reports and administrative information.

goods							
id	route					material	
	num	location	$ task		num	prod	real
			operation	capacity			
897654	1	’EindhovenDC’			2	’chassisX19’	1
	2	’ParisPU8’	’drillingFA8’	2.55		’wheelT45’	4
			’grindingDR7’	1.08		’engineFM11’	1
			’assembleRT6’	1.29			
	3	’LyonPU9’	’paintHG9’	4.93			
			’polishIR7’	0.08			
	4	’MadridDC’					

Table 5.4: A value of type `goods`

5.5.2 The supply system

The first building block we are going to describe is the `supply` system. The `supply` system is used to represent one or more suppliers taking care of the input of raw material and components into the logistic chain. Note that a `supply` system (partly) defines the scope of the logistic chain we want to consider (i.e. the system boundary), because in our library a `supply` system is the ‘source’ of materials. The header of the `supply` system is shown below:

```
sys supply[in request:request,
           out response:response, outgoods:goods,
           val location:location,
               expectedhandlingtime:real,
               acceptrule:(prod->((real><real)><conditions)),
               averagesupplydelay:(real><(prod->(real><real))),
               variancesupplydelay:real,
           fun supplydelay[mu:real,sigma:real,r:real]:real
          ]
```

The system has one input pin (`request`) to accept requests for material. There are two output pins: one to respond (`response`) and one to deliver the goods (`outgoods`). Note that we use the term ‘pin’ to refer to an input or output channel (place) of the system. The value parameters are used to specify the (unique) location of the `supply` system, the average expected order lead time, some acceptance rules and the supply delay. The average supply delay is a fixed value per delivery, and for each product a fixed delay and a variable delay per item, as specified by `averagesupplydelay`. The variance of the distribution of the supply delay is given by `variancesupplydelay`. The distribution of the supply delay is specified by a function parameter `supplydelay`. The arguments of this function are the average and variance (calculated using the value parameters) and a random number. This way it is possible to specify any kind of distribution. A request for goods is accepted if the requested material is available in the period specified by the `timewindow` field in the request. The value parameter `acceptrule` specifies for each product, the period length, the maximum quantity available in each period and the supply conditions.

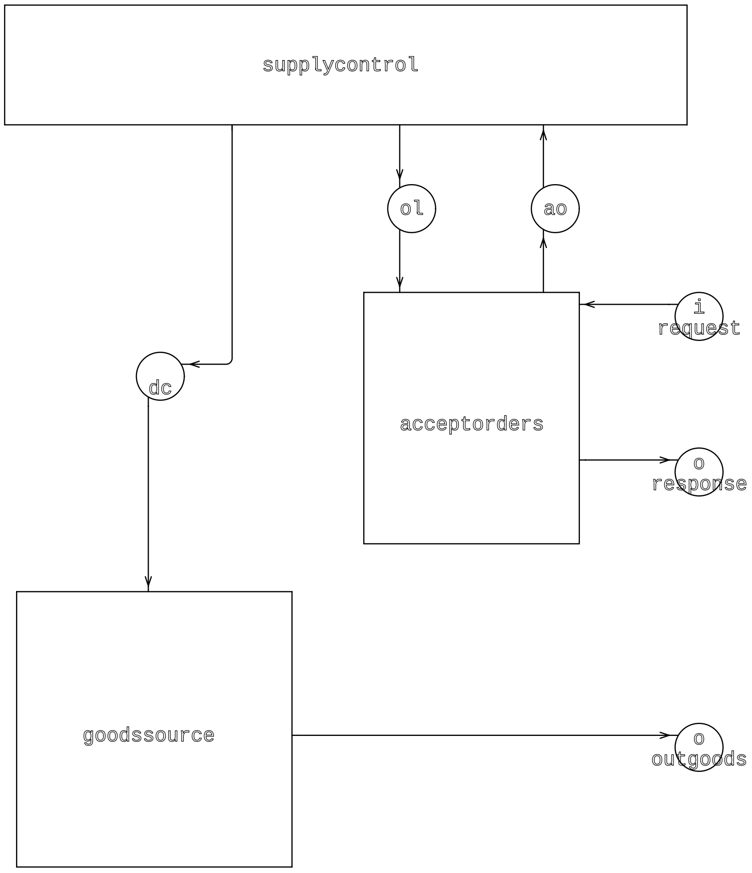


Figure 5.28: The supply system

If we zoom in, we see that a `supply` system consists of three subsystems, see figure 5.28.
The `goodssource` system takes care of the actual production (or a substitute) and the delivery of the materials requested. The header of this system is given below:

```
sys goodssource[in dc:delivercommand,
                out outgoods:goods,
                val averagesupplydelay:(real><(prod->(real><real))),
                    variancesupplydelay:real,
                fun supplydelay[mu:real,sigma:real,r:real]:real
               ]
```

Note that the delay distribution of a delivery is specified by the value and function parameters that have been discussed for the `supply` system.

The subsystem `acceptorders` handles the requests for materials:

```
sys acceptorders[in ol:orderlimit, request:request,
                 out response:response, ao:acceptedorder,
                 val location:location,
                     expectedhandlingtime:real
                ]
```

A request has an identification, a route, a list of material, a time window, a condition and a commit field. If the commit field is 'false', then the request is an inquiry without any obligations. However, if the commit field is 'true', then the request will be satisfied if possible and the requesting party is obliged to accept the corresponding material (or service). In both cases a response will follow having the same kind of attributes. If the commit field of the response is 'true', then the material will be delivered, in all likelihood within the timewindow, given the conditions requested. Note that such a response is only possible if the commit field in the request was 'true' and there are sufficient resources to satisfy the request. To estimate the orderleadtime, the `acceptorders` system uses the value parameter `expectedhandlingtime`. The input pin `ol` of type `orderlimit` specifies the ordervolume that can be accepted for each period satisfying some minimal conditions.

The physical elementary system (PES) `goodssource` and the information handling elementary system (IES) `acceptorders` are both controlled by the control system (CS) `supplycontrol`:

```
sys supplycontrol[in ao:acceptedorder,
                  out ol:orderlimit, dc:delivercommand,
                  val acceptrule:(prod->((real><real)><conditions))
                 ]
```

The value parameter `acceptrule` specifies for each product the conditions (for example the quality or price of the product) and the maximum quantity that can be delivered in each period (the other field of type `real` is used to denote the length of the time interval). This value parameter is used to produce tokens of type `orderlimit` to inform the `acceptorders` system about the maximal quantity that can be supplied.

5.5.3 The demand system

The demand for end-products is generated by a `demand` system. This system is a building block used to represent a class of customers. In a way the `demand` system is the complement of the `supply` system. This component defines the other end of the logistic chain we want to consider, because it is a 'sink' absorbing finished products. The header of the `demand` system is shown below:

```
sys demand[in response:response, ingoods:goods,
           out request:request,
           val location:location,
               suppliertable:(prod->((location->num)><conditions)),
               expectedorderleadtime:real,
               demand:(prod->((real><real)><(real><real))),
               requestedleadtime:real,
           fun interarrivaltime[mu:real,sigma:real,r:real,t:real]:real,
               orderquantity[mu:real,sigma:real,r:real,t:real]:real
          ]
```

There is one output pin to order goods (`request`), and two input pins, one to receive goods (`ingoods`) and one to be informed about the requests (`response`). The location of the demand system is specified by the `location` parameter. The parameter `suppliertable` is used to determine where to order a specific product. The demand process is specified by the value parameter `demand` and the function parameters `interarrivaltime` and `orderquantity`. For each product `demand` specifies the average and variance of the interarrival time (the time between two successive requests for the product) and the average and variance of the orderquantity. These figures are used to calculate actual interarrival time and orderquantity using the functions `interarrivaltime` and `orderquantity` respectively. Both may depend on a random number (`r`) and the current time (`t`). This way it is possible to model stochastic distributions and seasonal trends. The parameter `expectedorderleadtime` is the expected time it takes to deliver a requested product. The parameter `requestedleadtime` is the maximal time between the moment the demand exists and the moment a demand is satisfied. Figure 5.29 shows the internal structure of the `demand` system.

The system `goodssink` accepts goods for the `demand` system and reports every delivery to the `demand` system via the output pin `rs`:

```
sys goodssink[in ingoods:goods,
              out rs:receivesignal,
              val location:location
```

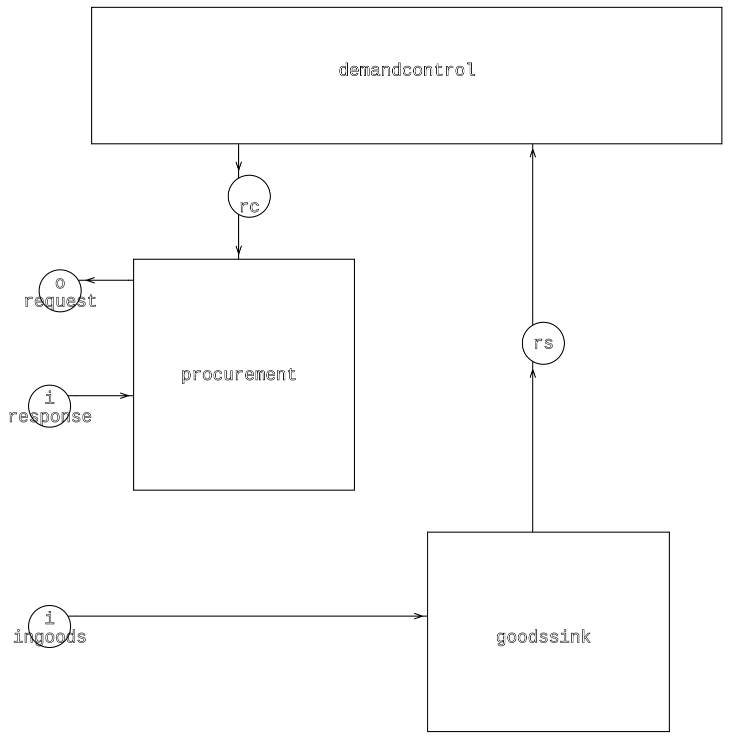


Figure 5.29: The demand system

```
            ]
```

The `demandcontrol` system generates the demand for products using the parameters `demand`, `interarrivaltime` and `orderquantity`. This results in a 'replenishment command'. The `timewindow` associated with the demand starts at the generated demand time and ends some time later, as defined by the parameter `requestedleadtime`.

```
sys demandcontrol[in rs:receivesignal,
                  out rc:replenishcommand,
                  val location:location,
                      demand:(prod->((real><real)><(real><real))),
                      requestedleadtime:real,
                  fun interarrivaltime[mu:real,sigma:real,
                                       r:real,t:real]:real,
                      orderquantity[mu:real,sigma:real,
                                    r:real,t:real]:real
                 ]
```

The `procurement` system has one output pin to order goods (`request`) and two input pins: one to receive information about a request (`response`) and one to accept replenishment commands (`rc`).

```
sys procurement[in rc:replenishcommand, response:response,
                out request:request,
                val location:location,
                    suppliertable:(prod->((location->num)><conditions)),
                    expectedorderleadtime:real
               ]
```

A replenishment command is a table specifying the demand for each product in a certain period. The `procurement` system tries to order these products using a strategy defined by the value parameter `suppliertable`. This table specifies for each product the minimal conditions (for example price or quality) the product has to satisfy and a preference list of suppliers (`location->num`). Note that in this context a supplier is a location able to deliver some products, for example a production unit, a distribution center or a supplier in a narrower sense (the `supply` component). The `procurement` system tries to order a product at the location with the highest preference. If there are several locations with the same preference, then an inquiry is done to find the best supplier (the commit field is 'false'). Otherwise, the inquiry is skipped and an order is sent to the supplier (the commit field is 'true'). If this first attempt does not give a supplier able to deliver the goods within the time window under the specified conditions, then the suppliers with the second best preference are consulted, etc. The value parameter `expectedorderleadtime` is used to time the requests.

5.5.4 The production unit

The `pu` system takes care of the transformation of products. One can think of a machine or a production unit. The header of the `pu` system is:

```
sys pu[in incommand:aggprodcommand, requestin:request,
          responsein:response, ingoods:goods,
       out outstatus:aggprodsignal, responseout:response,
           requestout:request, outgoods:goods,
       val bom:billofmaterial,
           reporttime:real,
           location:location,
           suppliertable:(prod->((location->num)><conditions)),
           expectedorderleadtime:real,
           expectedhandlingtime:real,
           initmeans:$means,
       fun producefunction[demand:(prod><timewindow)->real,
                              maxprodlevel:(prod><timewindow)->real,
                              inprocessinv:material,
                              freemeans:$means,
                              busymeans:(means->(real><material)),
                              bom:billofmaterial,
                              time:real
                             ]:($realtimecommand><$replenishcommand)
      ]
```

The input pin `ingoods` and the output pin `outgoods` represent the flow of goods between the production unit and its environment. If a production unit is unable to produce some material (for example raw materials), it tries to order these using the `requestout` and `responsein` pins. The actual demand for products is handled using the `requestin` and `responseout` pins. The pins `incommand` and `outstatus` are used to interact with some higher authority at the level of aggregated production plans. The time between two reports to this higher authority is specified by the value parameter `reporttime`. It is obvious that a production unit has a location (`location`), a list of suppliers (`suppliertable`) and a bill of material (`bom`). Furthermore, a production unit has a number a resources (`initmeans`). For the `pu` system we assume that the number of resources is constant, it is easy to extend this to a variable number of resources. The internal structure of the `pu` system is shown in figure 5.30.

One of the subsystems is the following physical elementary system:

```
sys transformer[in pc:realtimeprodcommand, ingoods:goods,
                   dc:delivercommand,
                out ps:realtimeprodsignal, outgoods:goods,
                val location:location,
                    initmeans:$means
               ]
```

This system receives commands of the type:

```
 type realtimeprodcommand from material >< means >< task >< material;
```

Such a command specifies a transformation process transforming some material into some other material by executing a task using some means. The `transformer` system also accepts goods arriving via the input pin `ingoods`. Finished goods leave the system via `outgoods`. The release of finished products is initiated by a 'deliver command' via the input pin `dc`. If a task has been executed, this is reported via the output pin `ps`.

The `realtimecontroller` system controls the `procurement`, `acceptorders` and the

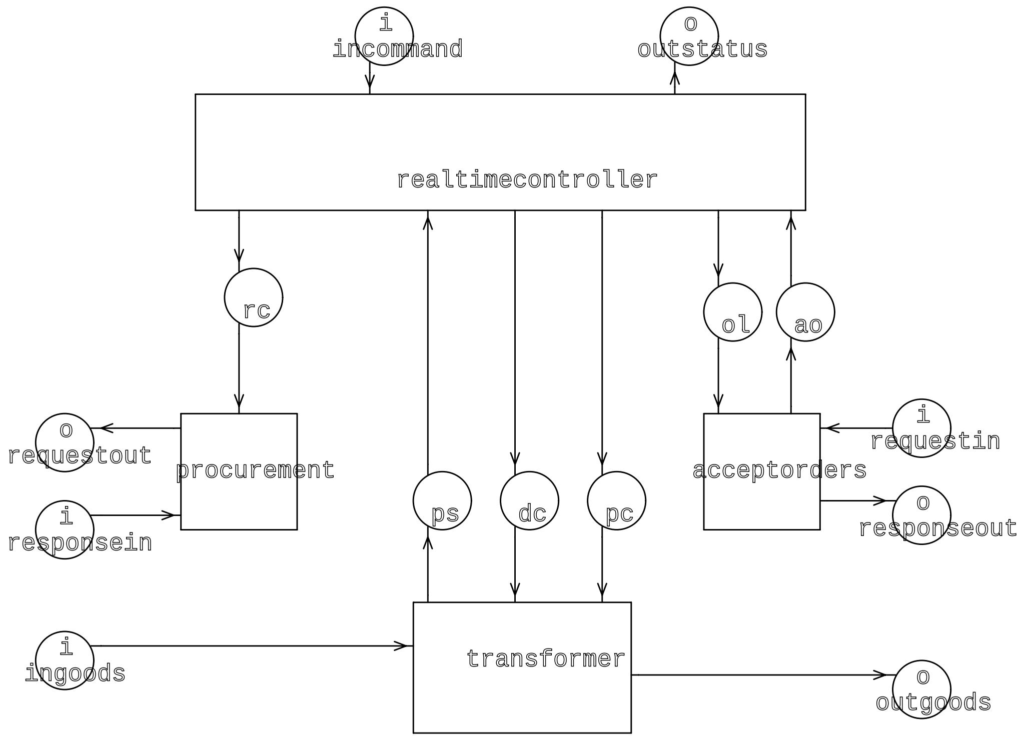


Figure 5.30: The production unit

`transformer` system:

```
sys realtimecontroller[in incommand:aggprodcommand,
                          ps:realtimeprodsignal,
                          oa:acceptedorder,
                       out outstatus:aggprodsignal,
                           pc:realtimeprodcommand,
                           rc:replenishcommand, ol:orderlimit,
                           dc:delivercommand
                       val bom:billofmaterial,
                           reporttime:real,
                       fun producefunction[
                               demand:(prod><timewindow)->real,
                               maxprodlevel:(prod><timewindow)->real,
                               inprocessinv:material,
                               freemeans:$means,
                               busymeans:(means->(real><material)),
                               bom:billofmaterial,
                               time:real
                                          ]:($realtimecommand ><
                                               $replenishcommand)
                      ]
```

billofmaterial				
prod	material		task	
	prod	real	operation	capacity
'finishedcarMB2'	'carMB2'	1.	'paintCS3'	0.2345
'carMB2'	'wheelF3'	4.	'assemble'	7.6435
	'chassisG1'	1.		
'bikeFX3'	'wheelH2'	2.	'assemble'	5.3645
	'chassisP1'	1.		

Table 5.5: A value of type `billofmaterial`

The controller receives commands via the input pin `incommand` of type:

```
type aggprodcommand from prod -> ((timewindow -> real) >< conditions);
```

This command specifies the maximum production levels for each period. The value parameter `reporttime` specifies the time between two successive reports. The behaviour of the `realtimecontroller` system is mainly specified by the function parameter `producefunction`. The `demand` parameter of this function represents the actual demand for each period, `maxprodlevel` gives the (maximum) production levels set by some higher authority, `inprocessinv` is the inprocess inventory, `freemeans` are the means ready to perform a task. The parameter `busymeans` represents the means that are performing a task, their expected termination time and the expected yield (`material`). The parameter `bom` specifies all production steps and is of type:

```
type billofmaterial from prod -> (material >< task);
```

Table 5.5 shows an example of such parameter.
If a product is not in the domain of the mapping, then it has to be ordered, i.e. the `realtimecontroller` system sends a 'replenishment command' to the `procurement` system. Note that the `producefunction` returns zero or more commands for both the `transformer` system and the `procurement` system. Using this function parameter it is possible to implement many production control methods (for example MRP). The `realtimecontroller` also controls the `acceptorders` system, it specifies the maximum ordervolume that can be accepted for each period.

The building block `pu` distinguishes between aggregate production planning and detailed (real-time) production control. Inside the production unit jobs are scheduled for a specific machine (means), the outside world is not aware of the existence of machines. The same holds for the intermediate products needed to produce an end-product, for example sub-assemblies. The products controlled by the outside world are the so-called *goods flow controlled items* (see Bertrand et al. [18]). Typical products to be controlled outside the `pu` system (i.e. via `incommand` and `outstatus`) are the MPS-items.

5.5.5 The stock point

In this section we describe a number of building blocks to model inventories. We start with the `sp` system, where `sp` stands for stock point. Examples of stock points are a regional warehouse, a distribution centre or a storage area containing supplies and raw materials. The main characteristic of our stock point is that it has a more or less autonomous behaviour. The header of the `sp` system is:

```
sys sp[in incommand:replenishmentstrategy, responsein:response,
          ingoods:goods, requestin:request,
       out outstatus:inventorylevels, requestout:request,
           outgoods:goods, responseout:response,
       val reporttime:real,
           location:location,
           suppliertable:(prod->((location->num)><conditions)),
           expectedorderleadtime:real,
           expectedhandlingtime:real,
       fun replenish[s:replenishmentstrategy,physicalstock:material,
                     demand:((prod><timewindow)->real),
                     ordered:((prod><timewindow)->real)
                    ]:replenishcommand,
           orderlimit[s:replenishmentstrategy,physicalstock:material,
                     demand:((prod><timewindow)->real),
                     ordered:((prod><timewindow)->real)
                     ]:orderlimit,
           handleintime[x:material]:real,
           handleouttime[x:material]:real
      ]
```

There are four input pins and four output pins. The pins `ingoods` and `outgoods` represent the flow of goods in and out of the stock point. If some external party needs some products, it sends a request to the stock point via the channel connected to `requestin`. The stock point responds via `responseout`. The main objective of a stock point is to keep inventories of certain products, if the inventory level of a product falls below a certain level or we want to anticipate on future developments, then a replenishment is needed. To order the products necessary for such a replenishment, we have the pins `requestout` and `responsein`. The replenishment strategy can be altered by some 'higher' authority via the `incommand` and `outstatus` pins. The meaning of the value and function parameters will be discussed when we describe the subsystems of sp shown in figure 5.31.

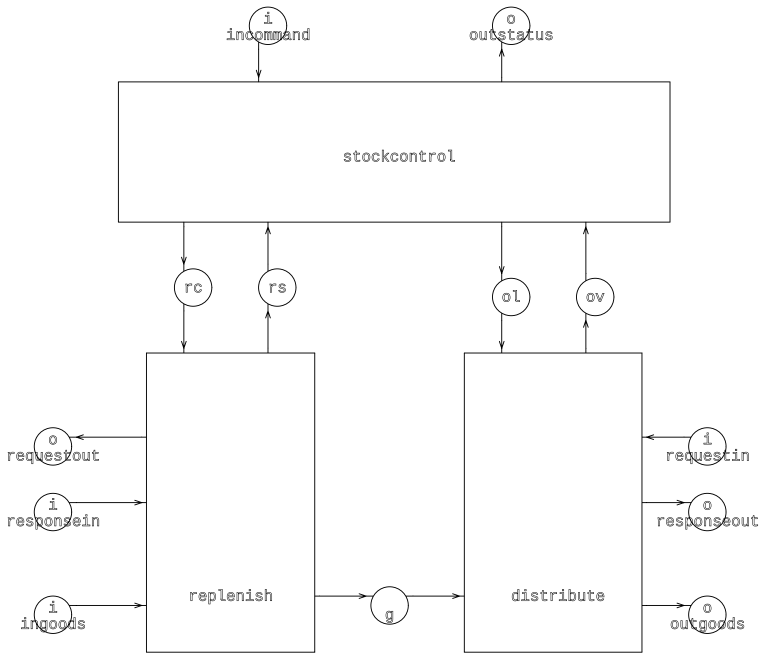


Figure 5.31: The stock point

The system `stockcontrol` controls the other two logistic subsystems `replenish` and `distribute`:

```
sys stockcontrol[in incommand:replenishmentstrategy,
                    rs:replenishsignal,
                    ov:ordervolume,
                 out outstatus:inventorylevels,
                     rc:replenishcommand,
                     ol:orderlimit,
                 val reporttime:real,
                 fun replenish[s:replenishmentstrategy,
                               physicalstock:material,
                               demand:((prod><timewindow)->real),
                               ordered:((prod><timewindow)->real)
                              ]:replenishcommand,
                     orderlimit[s:replenishmentstrategy,
                                physicalstock:material,
                                demand:((prod><timewindow)->real),
                                ordered:((prod><timewindow)->real)
```

```
                          ]:orderlimit
             ];
```

This system has an interface with some higher authority which tells the system to change its replenishment strategy. This strategy is defined for each product, see table 5.3. A strategy has a name and a number of parameters. Based on this strategy and the function parameter `replenish` the system issues replenishment commands via output pin `rc`. The parameters of the function `replenish` are the strategy (`s`), the current stock (`physicalstock`), the backorders and expected demand (`demand`) and the products already ordered (`ordered`). The input pin `rs` keeps the `stockcontrol` system informed about the (physical) replenishments. The output pin `ol` of type `orderlimit` is used to pass the upper bounds for the quantity of distributed goods in each period to the `distribute` system. Note that these maximum order quantities are calculated using the function parameter `orderlimit`. The parameters of this function are identical to the parameters of the `replenish` function. The input pin `ov` keeps the `stockcontrol` system informed about the physical stock (`material`) and the actual demand for products (`(prod><timewindow)->real`). From time to time the system reports the physical stock level, the demand level and the amount of ordered products using the output pin `outstatus`. The time between two successive reports is set using the `reporttime` parameter.

The system `replenish` takes care of the ordering of goods to replenish the stock:

```
sys replenish[in incommand:replenishcommand, response:response,
                 ingoods:goods,
              out outsignal:replenishsignal, request:request,
                  outgoods:goods,
              val reporttime:real,
                  location:location,
                  suppliertable:(prod->((location->num)><conditions)),
                  expectedorderleadtime:real
             ]
```

The meaning of the input and output pins follows directly from figure 5.31. The `replenish` system accepts all goods addressed to the `location` parameter and sends them to the channel connected to `outgoods`. Periodically, the total quantity of accepted goods is reported. The time between two successive reports is specified by the value parameter `reporttime`. The value parameters `suppliertable` and `expectedorderleadtime` are used to order the products.

The system `distribute` accepts orders, stores products and distributes them:

```
sys distribute[in incommand:orderlimit, request:request,
                  ingoods:goods,
               out outstatus:ordervolume, response:response,
                   outgoods:goods,
               val location:location,
                   reporttime:real,
                   expectedhandlingtime:real,
               fun handleintime[x:material]:real,
                   handleouttime[x:material]:real
              ]
```

The meaning of the pins is straightforward given figure 5.31. The `distribute` system reports the current inventory level and the accepted orders from time to time (as specified by `reporttime`) via the output pin `outstatus`. The value parameter `expectedhandlingtime` is used to determine whether it is possible to deliver within the requested time window. An upper bound for the number of products that can be supplied in each period is given via the input pin `incommand`. The two function parameters represent the time it takes to store and the time to pick some material.

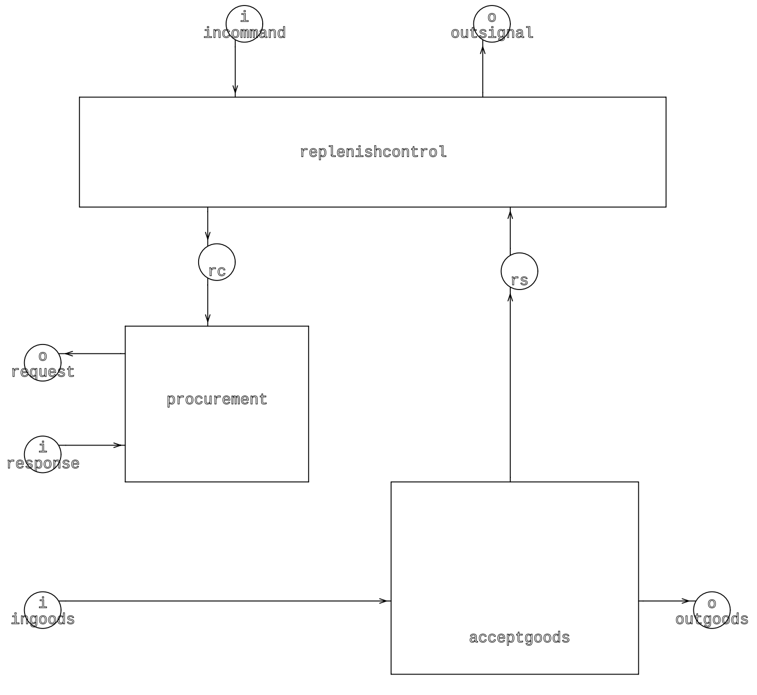


Figure 5.32: The replenish subsystem

Now it is time to take a closer look at the logistic subsystems `replenish` and distribute. Figure 5.32 shows the internal structure of the replenish system. It contains three subsystems: replenishcontrol, procurement and acceptgoods. The replenishcontrol system passes the replenishment commands to the procurement system and reports the total amount of received goods for each period.

```
sys replenishcontrol[in incommand:replenishcommand, rs:receivesignal,
                     out outsignal:replenishsignal, rc:replenishcommand,
                     val reporttime:real
                    ]
```

The value parameter reporttime is used to specify the time between two successive reports via outsignal. Every receipt of goods is reported by the acceptgoods system via the pin rs. The header of the acceptgoods system is:

```
sys acceptgoods[in ingoods:goods,
                out rs:receivesignal, outgoods:goods,
                val location:location
               ]
```

Note that the procurement system is also subsystem of demand and pu.

The internal structure of the distribute system is shown in figure 5.33. The subsystem acceptorders handles the incoming requests for goods and reports all accepted orders to the distributioncontrol system. Note that acceptorders was also used in the supply and pu system. The control system distributioncontrol passes the maximum order quantity for each period to the acceptorders system. It also controls the stockholding system by issuing commands via the output pin dc of type delivercommand.

```
sys distributioncontrol[in  incommand:orderlimit, ss:stocklevel,
                            ao:acceptedorder,
                        out outstatus:ordervolume, dc:delivercommand,
                            ol:orderlimit,
                        val reporttime:real,
                            expectedhandlingtime:real
                       ]
```

The parameter reporttime represents the time between two successive reports issued via the output pin outstatus. The parameter expectedhandlingtime is used to time the deliver commands to the stockholding system. The stockholding system sends updates of the actual stock level to the distributioncontrol system.

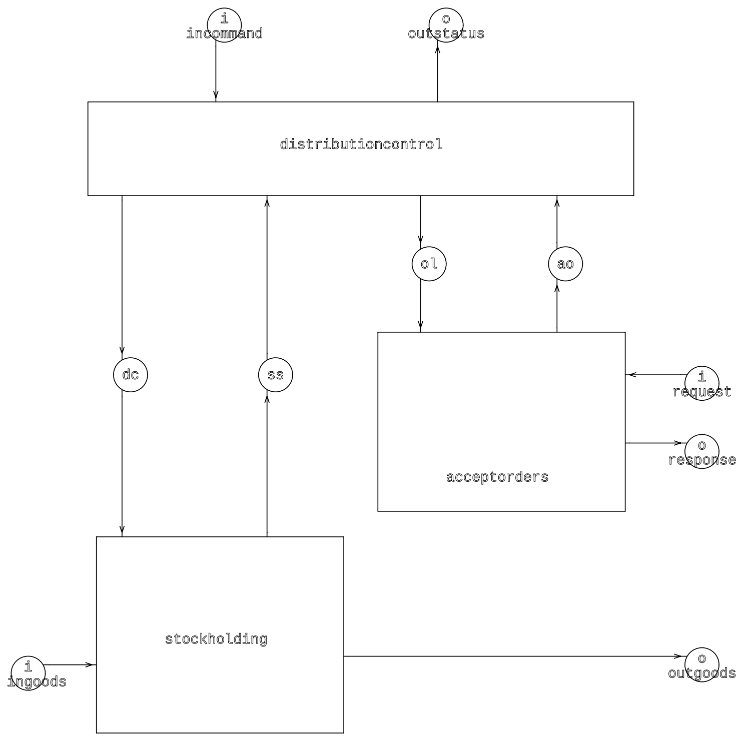


Figure 5.33: The distribute subsystem

The header of the `stockholding` system is:

```
sys stockholding[in dc:delivercommand, ingoods:goods,
                 out ss:stocklevel, outgoods:goods,
                 val location:location,
                 fun handleintime[x:material]:real,
                     handleouttime[x:material]:real
                ]
```

This system represents the physical warehousing process. The main activities are: accept goods, store goods and orderpicking. The time to store some material is given by the function parameter `handleintime`. The time it takes to fetch something is given by `handleouttime`.

5.5.6 The transport system

Finally, we discuss the building blocks associated with transport. In many cases it is sufficient to model transport by a 'delay'. For example, add the transport time to the `handleouttime` in the `sp` system. If we want to model transport in more detail, we can use the `transport` system. A typical example that can be modelled using this system is a transporter with a number of trucks. The header of the `transport` system is:

```
sys transport[in incommand:transportstrategy, response:response,
                 ingoods:goods,
             out outstatus:transportperformance, request:request,
                 outgoods:goods,
             val location:location,
                 productcharacteristics:(prod->(operation><capacity)),
                 transtable:((location><location)->(real><real)),
                 initmeans:(means->location),
             fun routescheduling[
                   s:transportstrategy,work:$acceptedtransorder,
                   free:(means->location),
                   busy:(means->(real><location)),
                   productcharacteristics:
                       (prod->(operation><capacity)),
                   transtable:((location><location)->(real><real))
                              ]:($routecommand><availabletranscap),
                 transtime[mu:real,sigma:real,r:real,t:real]:real
            ]
```

The input pin `ingoods` is used to collect goods for transport. The output pin `outgoods` is used to deliver the goods at the desired location. To accomplish this task, the system uses a set of transportation means (e.g. trucks). The `transport` system is triggered by requests for transport that arrive via the input pin `request`. The system replies to tell whether it is possible to execute the request (`response`). There is also an interface to interact with some higher level of control: the pins `incommand` and `outstatus`. A transport system has an address to send the requests to (`location`), and some initial distribution of means (`initmeans`). Every product has a number of characteristics specified by `productcharacteristics`. This parameter tells what kind of transport is needed (`operation`) and how many units of capacity it requires (`capacity`). Some units of capacity are: a cubic meter (space), kilogramme (weight) or pallets. The `transtable` parameter specifies the average and variance in the time needed to transport something from one location to the other. The time for loading or unloading is included. The function parameter `transtime` is used to calculate the actual transportation time. Note that this time is also based on a random variable (`r`) and the current time (`t`). The function parameter `routescheduling` is used to specify the control which may depend on the transport strategy set out by some higher authority. If we zoom in, we see three subsystems as shown in figure 5.34.

The system `transcontrol` schedules the transport activities and informs the sys-

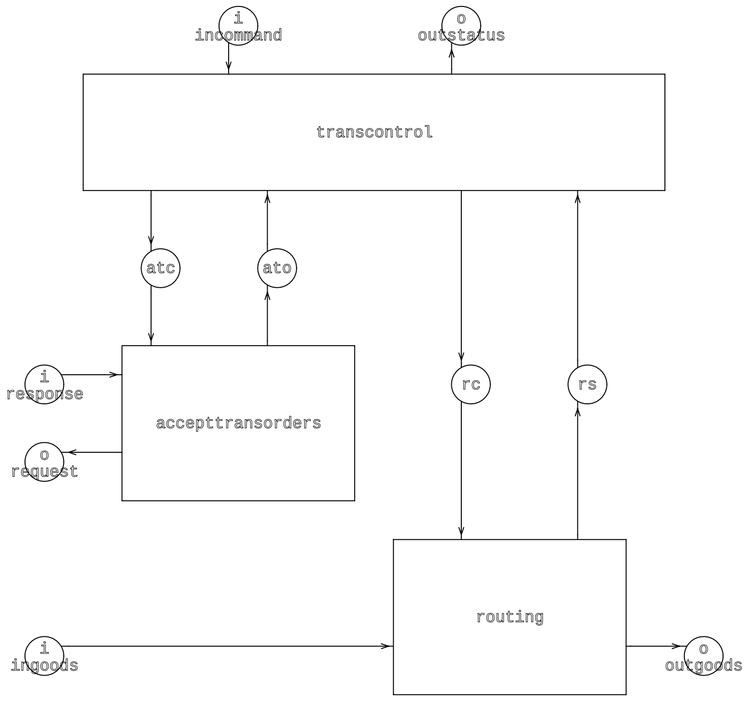


Figure 5.34: The transport system

tem `accepttransorders` about the (remaining) available transport capacity in each period. The header of the `transcontrol` system is:

```
sys transcontrol[in incommand:transportstrategy,
                    ato:acceptedtransorder,
                    rs:routesignal,
                 out outstatus:transportperformance,
                     atc:availabletranscap,
                     rc:routecommand,
                 val location:location,
                     productcharacteristics:
                         (prod->(operation><capacity)),
                     transtable:((location><location)->(real><real)),
                 fun routescheduling[
                       s:transportstrategy,work:$acceptedtransorder,
                       free:(means->location),
                       busy:(means->(real><location)),
                       productcharacteristics:(prod->(operation><capacity)),
                       transtable:((location><location)->(real><real))
                                   ]:($routecommand><
                                           availabletranscap)
                 ]
```

This system communicates about the aggregate control of the transport system with some higher authority using the pins `incommand` and `outstatus`. The pins `atc` and `ato` are used to interact with the `accepttransorders` system. The output pin `atc` is of type `availabletranscap` (see table 5.3), which specifies the remaining capacity for each operation and the conditions (for example price). In this context the term 'operation' refers to the specific kind of transport required. For example, liquid petrol gas and prefabricated chalets need different types of trucks. But it is possible for a truck to support different kinds of transport, see the definition of `means`. If a transport is accepted, then this is reported to the `transcontrol` system.

The pins `rc` and `rs` are used to communicate with the routing system. The type of `rc` is:

```
 type routecommand from (num -> (location >< $goods >< $goods)) ><
                        means;
```

Such a command specifies a list of locations (represented by a mapping) and the transport means involved. For each location the goods to collect and the goods to deliver are given. If a route is completed, then the `routing` system signals the location of the means (`rs`). The schedules for routing (`routecommand`) and the remaining transport capacity (`availabletranscap`) are calculated using the function parameter `routescheduling`. The parameters of this function are the strategy (`s`), the remaining set of accepted orders (`work`), the free and busy transport means (`free` and `busy`), the characteristics of every product (`productcharacteristics`) and the average and variance of the transporttime (`transtable`). Note that for all busy means the expected completion time and location of the corresponding route are given.

The `accepttransorders` system behaves similar to the `acceptorders` system. An order is accepted if there is sufficient capacity and the requested conditions are satisfiable.

```
sys accepttransorders[in atc:availabletranscap, response:response,
                      out ato:acceptedtransorder, request:request,
                      val location:location,
                          productcharacteristics:
                             (prod->(operation><capacity))
                     ]
```

The `routing` system takes care of the physical transport of goods. The actual transporttime is calculated on the basis of the parameters `transtable` and `transtime`.

```
sys routing[in ingoods:goods, rc:routecommand,
            out outgoods:goods, rs:routesignal,
            val location:location,
                initmeans:(means->location),
                transtable:((location><location)->(real><real)),
            fun transtime[mu:real,sigma:real,r:real,t:real]:real
           ]
```

The routing system accepts only those goods appearing in some 'route command'.

Note that is is not possible to connect systems such as pu, supply or sp directly to the transport system. Therefore our library contains the forwarder system. This system receives goods for transportation and forwards them to some transport system. The header of the forwarder system is:

```
sys forwarder[in ingoods:goods, response:response,
              out outgoods:goods, request:request
              val location:location,
                  transporterstable:(location->num)><conditions)
             ]
```

A detailed discussion of the internal structure of this system is beyond the scope of this chapter. The pins ingoods and outgoods represent the physical flow of products. The transporterstable is used to select the best transporter. The forwarder system tries to place a transport order at the location (transporter) with the highest preference. If there are several locations with the same preference, an inquiry is done to find the best transporter (the commit field is 'false'). Otherwise, the inquiry is omitted and a transport order is sent this transporter (the commit field is 'true'). If this first attempt does not give a transporter able to deliver the goods within the time window under the specified conditions, then the transporters with the second best preference are consulted, etc. The pins request and response are used to communicate with these transporters.

This concludes our description of the logistic library. We realize that this description is far from complete, but it gives the reader an impression of the modelling capabilities of such a library. Note that we did not describe the building blocks controlling a part of the logistic chain in an integral way (i.e. global control). We did not do this is because we think that the *structure* of such a control varies from case to case. Therefore, it is difficult to supply useful building blocks for this purpose. Moreover, the control decisions made at this level are often strategic. Strategic decision making is hard to model in a generic way.

5.6 Some guidelines

In the previous section we did not discuss a procedure for developing a model (or specification) in terms of the logistic components. However, to support the use of the logistic library, we also have to supply a method (see figure 5.27). The existence of such a method is of crucial importance. Without such a method, the components may be misused, thus yielding an erroneous or unnecessary complicated model (or specification).

The method we propose is made up of a number of guidelines which are partially based on the concepts developed in section 5.4. This is a direct consequence of the fact that these concepts have been used to develop the library.

Our method identifies a number of consecutive steps, when developing a model of a complex logistic system:

step 1 State the problem informally.

step 2 Identify the logistic parties involved.

step 3 Define the *system boundary* of the logistic system under consideration.

step 4 Decompose the system:

> **step 4a** If the system can be modelled by one of the components in the library, then replace the system by the corresponding component and proceed with step 5.
>
> **step 4b** If the logistic system is (1) a physical elementary system, (2) an information elementary system or (3) a control system, then describe the task and interactions of this system informally and proceed with step 5.
>
> **step 4c** Decompose the system into (1) a set of relatively independent logistic components, or (2) a set of logistic components controlled by a control system. For each of the subsystems proceed with step 4a.

step 5 Step 4 resulted in a hierarchical model composed of components and (undefined) physical elementary systems, information elementary systems and control systems. Install each component by instantiating the parameters with actual entities. Create a suitable system definition for each physical elementary system, information elementary system or control system which is not available in the library.

The development of a model has to start with the question: ‘Why do we want to model the logistic system?’. To answer this question, we have to state the problem properly. Based on this informal problem statement we identify the logistic parties involved, e.g. suppliers, consumers, transporters, etc. Given the relevant parties

involved, we determine the system boundary, i.e. the scope, of the logistic system under consideration.
Then we decompose the logistic system into subsystems, until each subsystem is a physical elementary system, an information elementary system, a control system or resembles a component.
For complex logistic systems, the decomposition hierarchy should be several layers deep. This to avoid a shallow hierarchy with systems composed of many subsystems. Note that step 4 represents an iterative process which balances between two objectives: (1) try to use as many existing components as possible and (2) the decomposition (hierarchy) has to as 'natural' as possible. If the library contains a lot of components, then it is difficult to find the appropriate component (or to determine that there is no appropriate component available). To support this task, it is necessary to develop tools for this purpose (e.g. a repository) and to educate the users of the library.
Although step 4c may raise the presumption that we advocate a 'pure' top down approach, it is also a bottom up approach, since we try to use existing components.

In a '80/20'-situation, 20 percent of the subsystems are physical elementary systems, information elementary systems and control systems, that have to be defined because no suitable component is available. For the construction of these system definitions we provide the following guidelines:

- abstract from irrelevant details
- decompose complex system definitions into suitable subsystems
- minimize the interfaces between subsystems
- parameterize the relevant characteristics
- use existing type definitions if possible (it increases the likelihood of a system being reusable)
- try to find a unifying system definition, when two or more systems, differ in terms of only a few aspects (this to avoid duplication)

If these guidelines and the five steps are followed closely, then the modelling process yields, in all likelihood, a satisfactory formal specification of the logistic system under consideration. In most cases, the main purpose of modelling is to prepare the system for analysis. There are several ways to analyse a system specified with ExSpect, see chapter 3.

Simulation is one of the most powerful analysis techniques to analyse a complex system. This brings us to the question: 'When is simulation useful?'. Many authors provide guidelines for answering this question (Shannon [112]). A possible reason for the application of simulation is the fact that analytical methods are unavailable or difficult to apply. An important advantage of simulation is that it helps the experimenter to understand and to gain a feel for the problem. For a more extensive description of the simulation process the reader is referred to Shannon [112] and Bratley et al. [25].

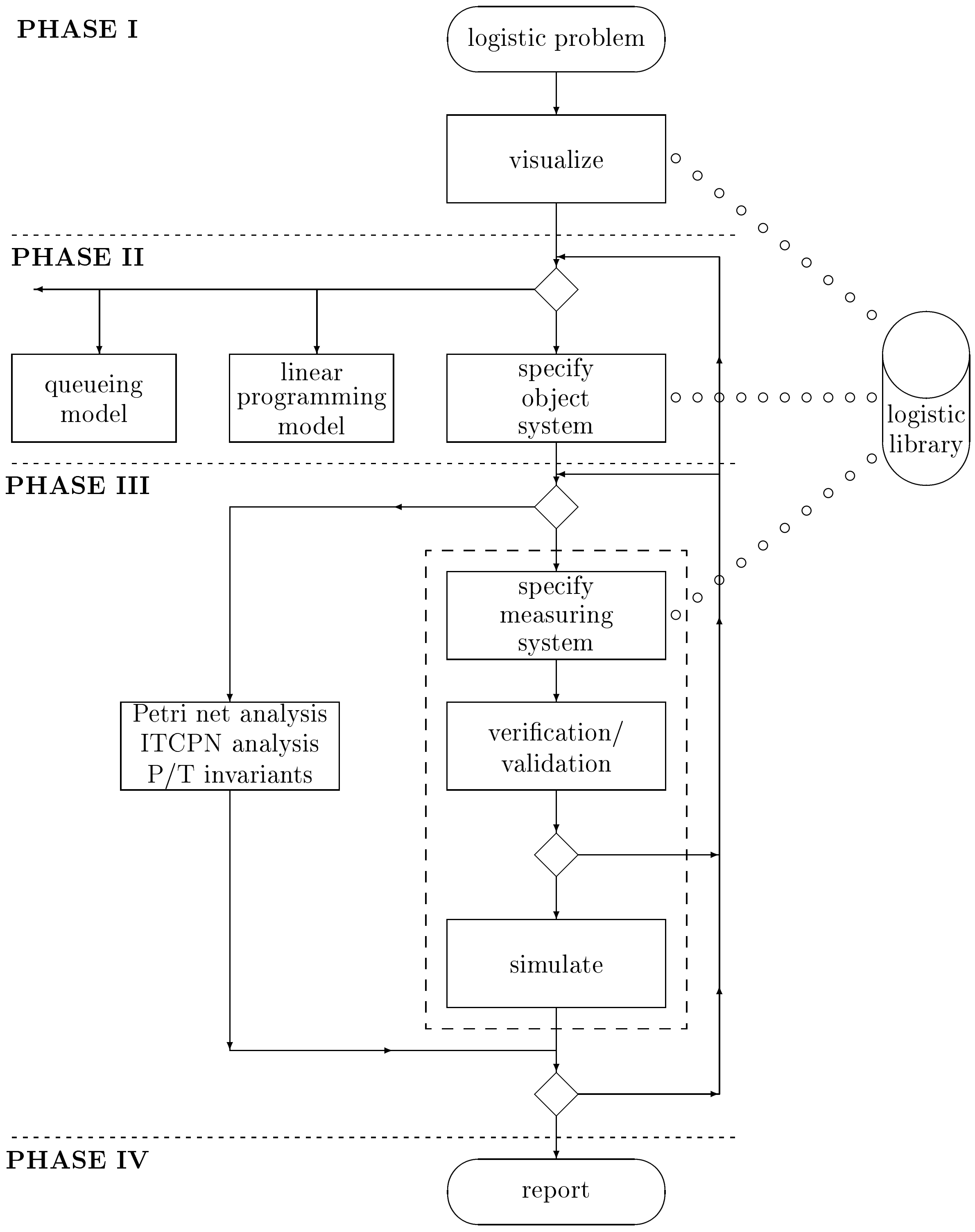


Figure 5.35: The phases in the modelling and analysis process of a complex logistic system

Assuming that simulation is likely to be used to analyse the logistic problem, we distinguish four phases in the modelling process. These phases are shown in figure 5.35.

In phase I we identify the logistic problem and 'visualize' the related logistic system. With 'visualize' we mean determining the system boundaries and a graphical description in terms of relevant components (building blocks). We also add an informal description of every component and the flows between these components. This visualizing process is an aid to thought and supports the communication between the modeller and the other people involved. Note that phase I corresponds to the modelling steps 1, 2, 3 and 4.

In phase II we have to decide whether we want to use the Petri net approach (ExSpect) or some other modelling or analysis technique. If the problem can be reduced to a simple model and solved analytically, there may be no need for simulation or the Petri net approach. Examples of models allowing for analytical solutions are queueing models and linear programming models. In all other cases we specify the logistic system in ExSpect. With specification we mean an unambiguous description of the model in terms of the ExSpect language. Note that this corresponds to modelling step 5. The specification process starts with the graphical description created in phase I. If possible, we use predefined building blocks from a logistic library. This saves a lot of effort.

In phase III we analyse the specification (of the logistic system) created in phase II. There are several ways to analyse such a Petri net based specification. Simulation is probably the most flexible way to analyse this specification. To simulate the system, we often have to specify a number of *measuring systems* and add them to the specification of the object system. A measuring system serves the purpose of presenting some results generated by the object system. Sometimes these measurements are incorporated in the logistic building blocks. Based on the information required, we also prepare the input data used by the simulation. Then we *verify* the model. This means that we check whether the specification operates in the way we think it does, that is, is the specification free of bugs and consistent with the informal model of phase I. Then the model is *validated*. Validation is the process that checks whether the specification is a sufficient close approximation of reality, for the intended application. If both tests succeed, we proceed with the actual experimentation. Otherwise, we go back to the specification process. Experimentation results in output data, that have to be interpreted. Based on this interpretation we adjust the parameters in the specification until we have obtained the desired results. Instead of simulation, we can also use a number of Petri net based analysis techniques, such as P and T-invariants and the ITCPN analysis techniques described in chapter 3.

In phase IV the specification and the results obtained are documented.

Although we identify four phases, in practice these phases will overlap and some iteration will be necessary (compare this with the well-known "Waterfall model" of software engineering [22]). For example, during the modelling phase we may start by modelling and simulating a simple system (to gain insight in the logistic problem situation). Then, via a number of iterations, the model is made more realistic. Because of the existence of the logistic library the number of iterations will be rather small. Although some iteration in the development of the specification is inevitable, we think it is useful to identify the phases shown in figure 5.35.

The ExSpect tool supports all phases. In phase I we can use the design interface of ExSpect to create a graphical description. In phase II we 'inherit' this description and use the design interface to create a complete specification. In this phase we also use the type checker to check the specification for correctness, consistency and completeness. In phase III we also use the interpreter and the runtime interface to simulate the specification. ExSpect also supports alternative analysis techniques: IAT allows for IT(C)PN analysis and the calculation of P and T invariants. Phase IV is supported by the possibility to add comments to the specification, the possibility to use graphical descriptions generated by the design interface and some export facilities to export data generated by the runtime interface.

Note that the phases I, II and III rely heavily upon the availability of a logistic library (see figure 5.35).

5.7 An example

To illustrate the use of the logistic library, we give an example of a logistic system modelled in terms of the building blocks described in the previous section. This case deals with a logistic system stretching out over the logistic chain from supplier to consumer. To keep the case description manageable and easy to comprehend, we use a fictitious example. Furthermore, our treatment of this example is intentionally abstract, e.g. we use symbolic names for products ($\mathcal{A}$,$\mathcal{B}$, ..) and locations ($S1$, $S2$, PU, ..). Nevertheless, we think this example illustrates the approach presented in this chapter.

5.7.1 The present situation

The structure of the logistic system is shown in figure 5.36. The company under consideration comprises two distribution centres ($SP1$ and $SP2$) and one manufacturing site (PU).
The two distribution centres hold inventory to supply a number of retailers. Every retailer is assigned to only one of the two distribution centres. These assignments are based on geographical motives.
The set of retailers assigned to distribution centre $SP1$ is denoted by $C1$, the set of retailers assigned to distribution centre $SP2$ is denoted by $C2$. The inventory

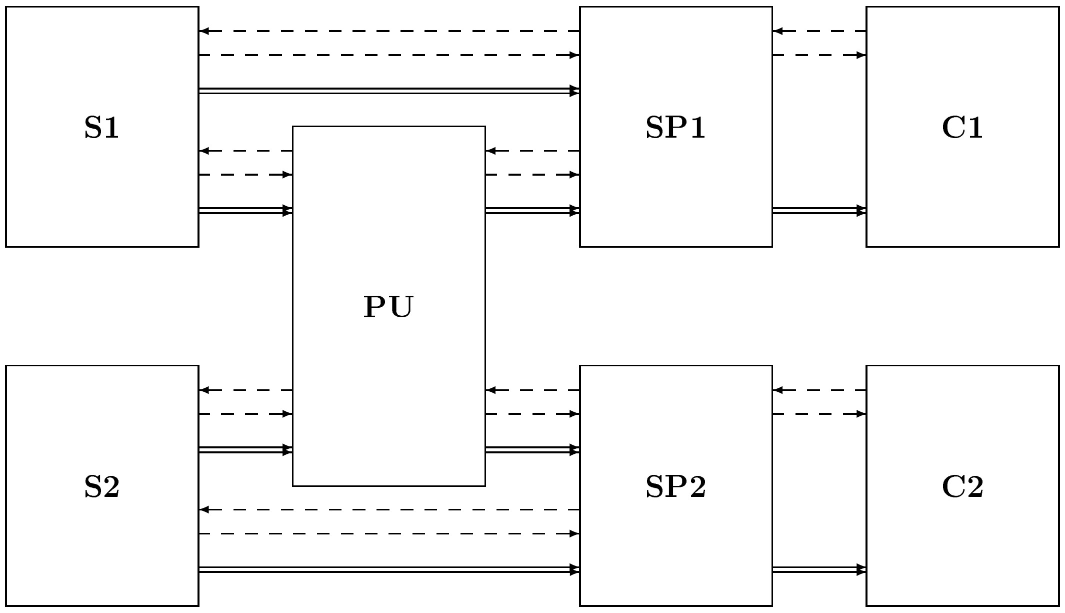


Figure 5.36: A logistic system

assortments of the distribution centres, i.e. the kind of products accumulated, are slightly different. This is caused by regional variations of demand.

The inventory of $SP1$ consists of products manufactured by PU and products purchased from supplier $S1$. Distribution centre $SP2$ acquires products from PU and supplier $S2$.

Manufacturing site PU produces the products which are not supplied by the two suppliers $S1$ and $S2$. To produce these products, PU acquires products from both suppliers. In the present situation, PU produces to order.

Given this informal description of the logistic chain, we will characterize the individual subsystems in terms of the logistic building blocks outlined in section 5.5.

The demand process

The set of retailers assigned to $SP1$ is represented by a `supply` building block. These retailers are allowed to order once a day. In this example there are only four kinds of products ordered by these retailers: $\mathcal{A}$, $\mathcal{B}$, $\mathcal{I}$ and $\mathcal{J}$. The number of items ordered fluctuates. In this case, the required quantity of each product is given a normal distribution, with the parameters specified in table 5.6.

The retailers assigned to $SP2$ are also represented by a `supply` building block. Instead of once a day, these retailers are allowed to order twice a day. A description of the demand generated by these retailers is given in table 5.7.

product	average	variance
$\mathcal{A}$	100	40
$\mathcal{B}$	200	80
$\mathcal{I}$	50	20
$\mathcal{J}$	60	24

Table 5.6: The quantity ordered by the retailers represented by $C1$ (daily = 24 hours)

product	average	variance
$\mathcal{A}$	75	30
$\mathcal{C}$	50	20
$\mathcal{D}$	100	40
$\mathcal{I}$	35	14
$\mathcal{K}$	40	16

Table 5.7: The quantity ordered by the retailers represented by $C2$ (twice a day = 12 hours)

The distribution centres

The main function of the distribution centres is to maintain inventories for the purpose of bringing the products near the customers and coordinating supply and demand. The distribution centres are represented by two `sp` systems.
Both stock points use a replenishment strategy where the inventory levels are checked twice a week. If the stock is below a certain level, a replenishment order is issued, the ordered quantity depends on the current stock. As a matter of fact, the ordered quantity is the difference between a predefined maximum level and the current operating stock. The corresponding values are given in table 5.8 and table 5.9. For example, if the current stock of product $\mathcal{K}$ in $SP2$ is 230 units, then 170 ($= 400-230$) units of product $\mathcal{K}$ are ordered ($230 < 240$).

product	minimum	maximum
	level	level
$\mathcal{A}$	300	500
$\mathcal{B}$	600	1000
$\mathcal{I}$	150	250
$\mathcal{J}$	180	300

Table 5.8: The minimum and maximum inventory levels of distribution centre $SP1$

product	minimum level	maximum level
$\mathcal{A}$	450	750
$\mathcal{C}$	300	500
$\mathcal{D}$	600	1200
$\mathcal{I}$	210	350
$\mathcal{K}$	240	400

Table 5.9: The minimum and maximum inventory levels of distribution centre $SP2$

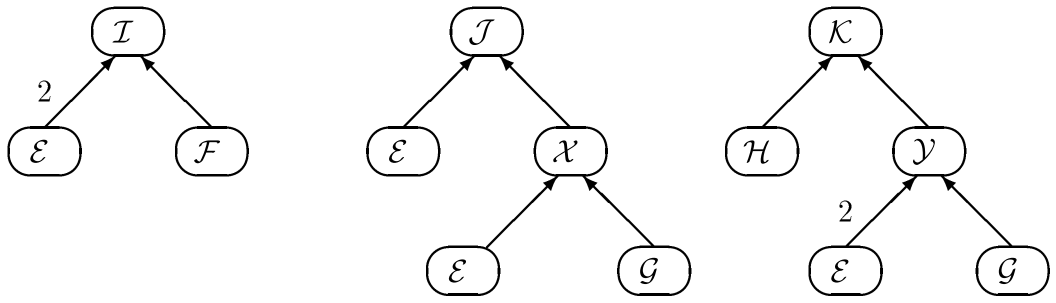


Figure 5.37: The bill of material of end-products $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$

Products $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$ are acquired from the manufacturing site PU. The other products are purchased from the local supplier, i.e. $SP1$ obtains products $\mathcal{A}$ and $\mathcal{B}$ from $S1$ and $SP2$ obtains products $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{D}$ from $S2$.

The production process

Manufacturing site PU is represented by a `pu` component. The PU produces the end-products $\mathcal{I}$, $\mathcal{J}$ and $\mathcal{K}$ from raw materials $\mathcal{E}$, $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$. Figure 5.37 shows the three bills of material. Note that there are two intermediate products $\mathcal{X}$ and $\mathcal{Y}$.

The transformations specified in figure 5.37, are performed by three capacity resources: 9901, 9902 and 9903. Capacity resource 9901 assembles two items $\mathcal{E}$ and one item $\mathcal{F}$ into one end-product $\mathcal{I}$. Resource 9902 assembles $\mathcal{E}$ and $\mathcal{X}$ into $\mathcal{J}$, and $\mathcal{H}$ and $\mathcal{Y}$ into $\mathcal{K}$. The subassemblies are performed by resource 9903. In the present situation, the production unit uses a 'MRP-like' production planning driven by actual demand, i.e. given the bill of material and the actual demand, the PU 'explodes' the requirements into a production and purchase schedule. This is specified by the function parameter `producefunction`. The required raw materials are purchased from suppliers $S1$ and $S2$. $S1$ supplies $\mathcal{E}$ and $\mathcal{F}$, $S2$ supplies $\mathcal{G}$ and $\mathcal{H}$. However, if supplier $S1$ is unable to supply product $\mathcal{E}$ within the given time window, then the PU sends a request to supplier $S2$.

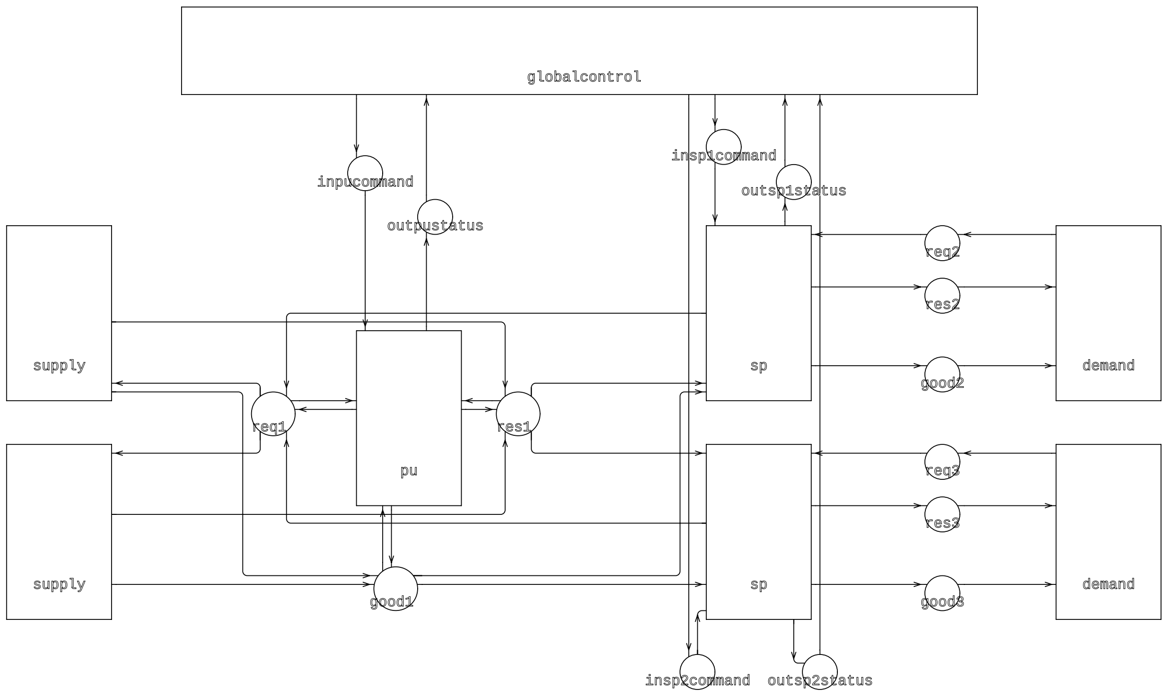


Figure 5.38: The logistic system in terms of the building blocks

The supply process

There are two suppliers $S1$ and $S2$, each represented by a `supply` system. $S1$ supplies $\mathcal{A}$, $\mathcal{B}$, $\mathcal{E}$ and $\mathcal{F}$, $S1$ supplies $\mathcal{A}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{E}$, $\mathcal{G}$ and $\mathcal{H}$. The maximum quantity and the time it takes to supply these products are given by a number of parameters.

Figure 5.38 shows the logistic system in terms of the logistic building blocks. A close observation shows that this figure resembles figure 5.36. We added the `glo-balcontrol` system to initialize the strategies used by the `pu` and `sp` systems. We did not model the transport between the locations explicitly. In this example, we assume that these transportation activities can be characterized by a stochastic delay distribution. These transport delays are added to the internal handling time of the building blocks. This completes our brief description of the present situation.

The description of the logistic system, just given, contains not nearly enough information to specify the parameters of the components. For example, to install the `sp` system, we have to supply a precise description of the replenishment strategy, the maximum order quantities, the time required to store and retrieve products from the warehouse, a detailed list of suppliers, etc. We have specified all of these parameters, this takes a few hours provided that the required data is available. Once this has been done, we can analyse the system.

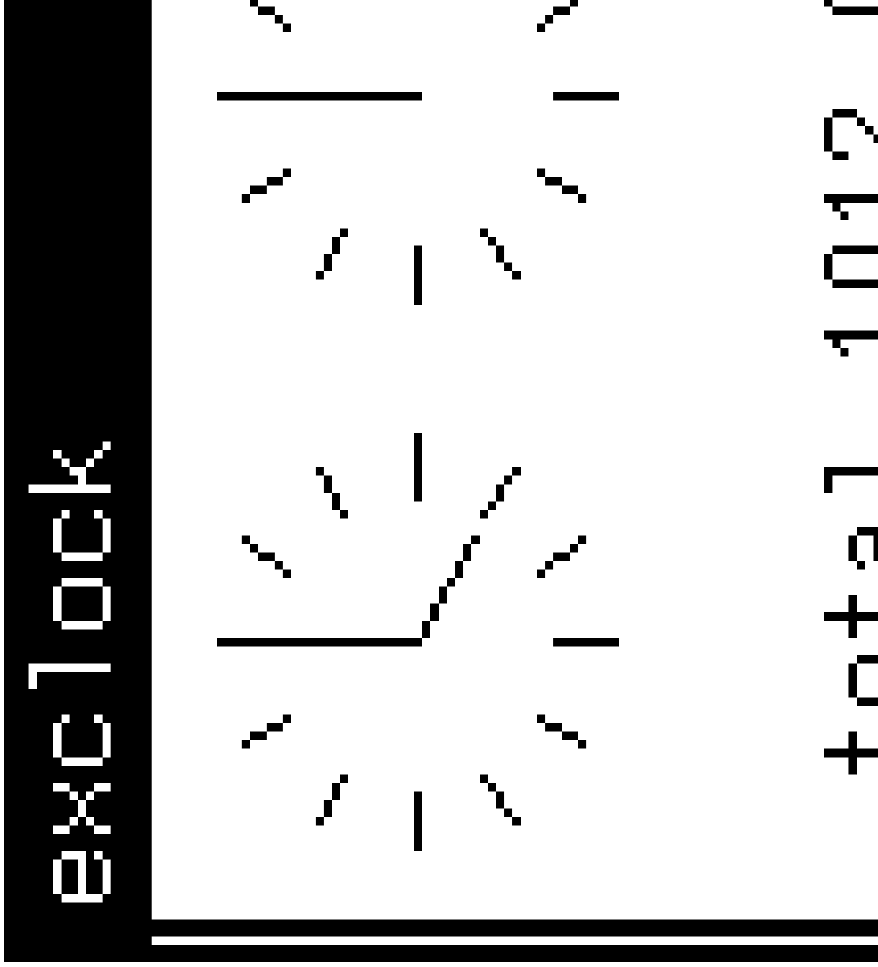


Figure 5.39: A screendump of a simulation

If we use simulation to analyse the system shown in figure 5.38, then several reports are presented. This is due to the fact that the logistic building blocks calculate several performance measures, e.g. order lead times, (average) stock levels, occupation rates, etc. Figure 5.39 shows a running simulation.

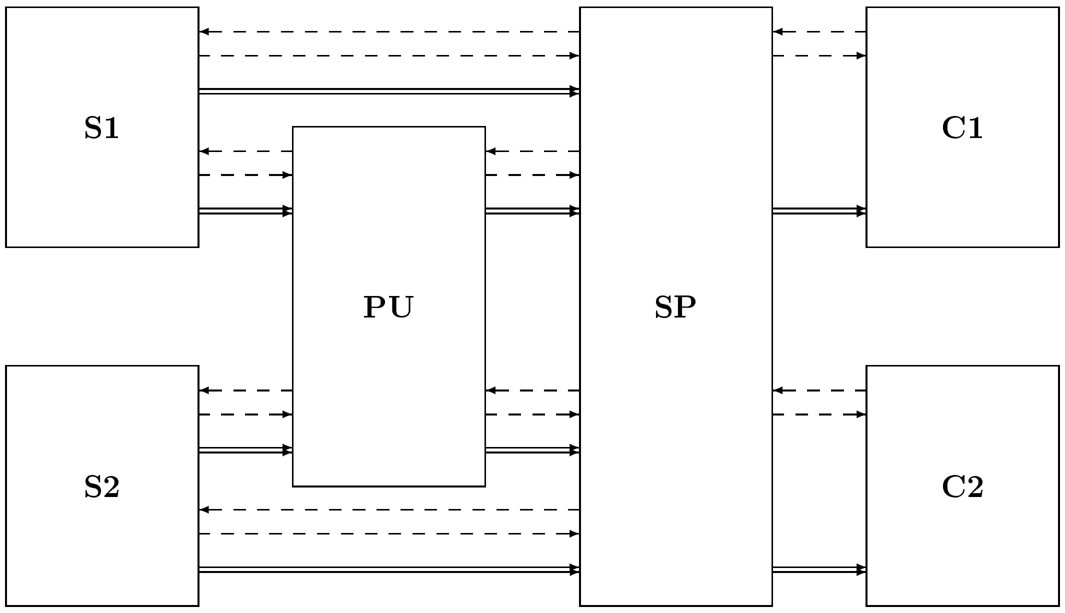


Figure 5.40: A logistic system with an alternative distribution structure

5.7.2 Alternatives

Given a specification of the present situation, it is easy to experiment with different alternatives, for example, another replenishment strategy or production based on forecasts instead of actual demand. It is also possible to analyse an alternative distribution structure. In figure 5.40, we show a situation with only one distribution centre.

5.8 Conclusion

We have developed a systematic approach to the modelling of logistic systems. This approach is based on our 'systems view' of logistics, described in section 5.4.
This view on logistics starts from the principle that any logistic system is composed of only three kinds of elementary systems: physical elementary systems, information elementary systems and control systems. Moreover, we identify typical relationships between these systems, i.e. we supply a taxonomy of the flows inside a logistic system.

Based on this ‘systems view’, we have developed a logistic library. The components in this library are highly generic and allow for the modelling of many logistic systems in a very ‘natural’ manner.

Our approach uses a framework based on a timed coloured Petri net model. Therefore, we investigated which role the theory, tools and methods described in the previous chapters can play in logistics. It turns out that timed coloured Petri nets are appropriate for the modelling of discrete logistic processes, because these nets allow for a graphical representation which is close to our intuition. Moreover, Petri nets have a firm mathematical foundation and allow for all sorts of analysis.

Although some elements of our framework are rather immature (e.g. the logistic library), experience shows that our approach is quite useful for the modelling and analysis of complex logistic systems.

Chapter 6

Conclusions and further research

The framework described in this monograph has been developed to solve problems related to the design and analysis of complex discrete dynamic systems. Although the emphasis is on logistics, most of the techniques and concepts described in the chapters 2, 3 and 4, also apply to other application domains, e.g. flexible manufacturing systems, distributed information systems and real-time systems.
The framework we propose is based on Petri nets and consists of:

- a timed coloured Petri net model
- a number of analysis methods
- a software package to create, modify and analyse timed coloured Petri nets
- a systems view of logistics
- a library of predefined logistic components

The systems we are interested in are often physically distributed and composed of many interacting components. Consider for example a typical logistic system made up of production units, stock points and transportation devices. Such a system is characterized by a continual exchange of goods, means and information.
Petri nets are appropriate for the modelling of these distributed systems, since they allow for the representation of parallelism and synchronization. However, Petri nets describing real systems tend to be complex and extremely large. Sometimes, it is even impossible to model the state space or the temporal behaviour of a system. To solve these problems we have developed the interval timed coloured Petri net (ITCPN) model described in chapter 2.
This model uses a new timing mechanism where time is associated with tokens and transitions determine a delay specified by an interval. The formal semantics of the ITCPN model have been defined by means of transition systems. The fact that time is in tokens results in transparent semantics and a compact state representation. Specifying each delay by an interval rather than a deterministic value or stochastic variable is promising, since it is possible to model uncertainty without having to bother about the delay distribution.

From the analysis point of view, the ITCPN model is also interesting, since interval timing allows for new analysis methods. In this monograph three analysis methods have been described.

The ATCFN method distinguishes itself by its simplicity. Although the ATCFN method has a number of serious drawbacks, it can be used in the field of project engineering.

The PNRT method can be used to analyse a larger, but still limited, set of ITCPNs (marked graphs satisfying some additional constraints). Many systems have been modelled using this type of nets. Typical application areas are flexible manufacturing and repetitive production scheduling. The PNRT method is reasonably efficient and answers questions about the arrival time of tokens (i.e. $\mathcal{EAT}_n(s,p)$ and $\mathcal{LAT}_n(s,p)$).

The MTSRT method is much more powerful, since it can be applied to arbitrary nets and answers a large variety of questions. This method constructs a reduced reachability graph. In such a graph a node corresponds to a set of (similar) states, instead of a single state. Although the MTSRT method performs a number of significant reductions, this graph may become too large to analyse. This is the reason we proposed two approaches to deal with this problem (see section 3.5). Another problem is the fact that the answers produced by the MTSRT method are not always as 'tight' as possible, because of dependencies between tokens. This is not a real handicap, since the results obtained by the MTSRT method are always valid and experimentation shows that, in general, these results are also meaningful.

The practical use of the ITCPN model and the three analysis methods depends to a large extent upon the availability of adequate computer tools. We use the software package ExSpect to create, modify and analyse our models. The design interface of ExSpect allows for the construction of models in a graphical manner. ExSpect supports three kinds of analysis: simulation, 'structural analysis' (invariants) and 'interval analysis' (MTSRT, PNRT, ATCFN). The availability of multiple kinds of analysis is a major advantage over other software packages.

We showed that the ITCPN model and the support offered by ExSpect are quite suitable for the modelling and analysis of logistic systems. However, the modelling of complex logistic systems is still a complicated task. This is the reason we presented a 'systems view of logistics', which is an attempt to structure the logistic domain. Based on a taxonomy of the flows in a logistic system, we have developed a systematic approach to the modelling of large and complex logistic systems. Insight into the interaction structure of a logistic system is vital for the effectiveness of the modelling process, because it supports the decomposition of the system into subsystems which are easier to understand. Our approach is intentionally abstract and starts from an idealized perception of the logistic domain.

Based on this systems view of logistics we have developed a small logistic library. During the development of this library we experienced the fact that a systems view of logistics facilitates the identification and creation of powerful building blocks. Although this library is rather immature, it shows that it is possible to attain a '80/20'-situation, i.e. a situation where 80 percent of the components needed are already available in a logistic library and take up only 20 percent of your time. But the 20 percent you have to create yourself take up 80 percent of your time. This implies that such a library increases the efficiency of the modelling process. Furthermore, the result of the modelling process is more succinct, more manageable and well-structured.

The framework presented in this monograph ranges from a method to model logistic systems to sophisticated, Petri net based, analysis methods. Some elements of this framework are quite mature whereas others raise new questions. These questions point out directions for continued research.

A direction of further research is the development of analysis methods based on interval timing. In this monograph we discussed three analysis methods (ATCFN, MTSRT and PNRT). It is quite possible that a number of existing methods for the analysis of (deterministic) timed Petri nets may be extended to our ITCPN model. Consider for example our SSPAT method presented in [2], which is a generalization of the analysis method described by Ramamoorthy and Ho in [107].

It is also possible to modify the MTSRT method such that the reduced reachability graph becomes smaller while sacrificing the tightness of the calculated bounds. For example, it is possible to aggregate 'similar' nodes in the reduced reachability graph into one 'super-node'. This super-node represents at least all the states represented by the 'old' nodes. Note that there is a trade-off between the computational efficiency and the strictness of the calculated results.

Another direction for further research is the addition of other types of analysis, e.g. Markovian analysis, queueing networks, perturbation analysis, etc. It is also possible to add other Petri net based analysis techniques, e.g. techniques to detect siphons and traps. Note that the ExSpect specification (or the ITCPN) is used as a 'blueprint' of the system under consideration. This blueprint can be observed from many angles and allows for various kinds of analysis. This is very convenient, since it prevents us from having to remodel the system every time we want to use an alternative analysis method.

Another item for further research is the development of a comprehensive reference model of logistics, based on our systems view of logistics described in chapter 5. This reference model should be validated by domain experts, i.e. logisticians as well as experts in operations research, control theory and industrial engineering. Such a reference model would give a fresh insight into the control of logistic systems. Furthermore, it would support the design of new logistic systems.

This reference model should be used to develop a new and extensive library of logistic building blocks. Without doubt, this library would increase the productivity of the modelling process. Furthermore, such a library would facilitate the diffusion of the logistic knowledge stored in the building blocks by domain experts.

Bibliography

[1] W.M.P. van der Aalst, *Specificatie en Simulatie met behulp van ExSpect* (in Dutch), Master's thesis, Eindhoven University of Technology, Eindhoven, 1988.

[2] ——, *Interval Timed Petri Nets and their analysis.* Computing Science Notes 91/09, Eindhoven University of Technology, Eindhoven, 1991.

[3] ——, *The modelling and analysis of queueing systems with QNM-ExSpect.* Computing Science Notes 91/33, Eindhoven University of Technology, Eindhoven, 1991.

[4] ——, *Logistics: a Systems Oriented Approach*, in Proceedings of the third International Working Conference on Dynamic Modelling of Information Systems, Noordwijkerhout, the Netherlands, June 1992, pp. 169–189.

[5] ——, *Modelling and Analysis of Complex Logistic Systems*, in Proceedings of the IFIP WG 5.7 Working Conference on Integration in Production Management Systems, Eindhoven, the Netherlands, 1992, pp. 203–218.

[6] W.M.P. van der Aalst, M. Voorhoeve, and A.W. Waltmans, *The TASTE project*, in Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, Bonn, June 1989, pp. 371–372.

[7] W.M.P. van der Aalst and A.W. Waltmans, *Modelling Flexible Manufacturing Systems with EXSPECT*, in Proceedings of the 1990 European Simulation Multiconference, B. Schmidt, ed., Nürnberg, June 1990, Simulation Councils Inc., pp. 330–338.

[8] ——, *Modelling logistic systems with EXSPECT*, in Dynamic Modelling of Information Systems, H.G. Sol and K.M. van Hee, eds., Elsevier Science Publishers, Amsterdam, 1991, pp. 269–288.

[9] W.M.P. van der Aalst, L. Somers, M. Voorhoeve, A.W. Waltmans et al., *ExSpect 3.0 User Manual*, Eindhoven, 1991.

[10] T. Agerwala, *Putting Petri Nets to Work*, IEEE Computer, 12 (1979), pp. 85–94.

[11] R.N. Anthony, *Planning and Control Systems: a framework for analysis*, Studies in management control, Harvard University, Graduate School of Business Administration, Boston, 1965.

[12] J.C.M. Baeten and J.A. Bergstra, *Real Time Process Algebra*, Formal Aspects of Computing, 3 (1991), pp. 142–188.

[13] F. Baskett, K.M. Chandy, R.R. Muntz, and F.G. Palacios, *Open, Closed and Mixed Networks of Queues with Different Classes of Customers*, Journal of the Association of Computing Machinery, 22 (1975), pp. 248–260.

[14] J.F.A.K van Benthem, *The Logic of time*, D. Reidel Publishing Company, Dordrecht, 1983.

[15] J.A. Bergstra and J.W. Klop, *Process Algebra for Synchronous Communication*, Information and Control, 60 (1984), pp. 109–137.

[16] B. Berthomieu and M. Diaz, *Modelling and verification of time dependent systems using Time Petri Nets*, IEEE Transactions on Software Engineering, 17 (1991), pp. 259–273.

[17] B. Berthomieu and M. Menasche, *An enumerative approach for analyzing time Petri nets*, in Information Processing: proceedings of the IFIP congress 1983, R.E.A. Mason, ed., vol. 9 of IFIP congress series, Elsevier Science Publishers, Amsterdam, 1983, pp. 41–46.

[18] J.W.M. Bertrand, J.C. Wortmann, and J. Wijngaard, *Production control: a structural and design oriented approach*, vol. 11 of Manufacturing Research and Technology, Elsevier Science Publishers, Amsterdam, 1990.

[19] F.P.M. Biemans, *Manufacturing Planning and Control: a reference model*, vol. 10 of Manufacturing Research and Technology, Elsevier Science Publishers, Amsterdam, 1990.

[20] F.P.M. Biemans and P. Blonk, *On the Formal Specification and Verification of CIM Architectures Using LOTOS*, Computers in Industry, 7 (1986), pp. 491–504.

[21] F.P.M. Biemans and C.A. Vissers, *Reference model for Manufacturing Planning and Control Systems*, Journal of Manufacturing Systems, 8 (1989), pp. 35–46.

[22] B.W. Boehm, *Software Engineering Economics*, Prentice-Hall, Englewood Cliffs, 1981.

[23] T. Bolognesi, F. Lucidi, and S. Trigila, *From Timed Petri Nets to Timed LOTOS*, in Proceedings of the IFIP WG 6.1 Tenth International Symposium on Protocol Specification, Testing and Verification (Ottawa 1990),

L. Logrippo, R.L. Probert, and H. Ural, eds., North-Holland, Amsterdam, 1990, pp. 1–14.

[24] D.J. BOWERSOX, *Logistical Management*, MacMillan, New York, 1974.

[25] P. BRATLEY, B.L. FOX, AND L.E. SCHRAGE, *A guide to simulation*, Springer-Verlag, New York, 1983.

[26] E. BRINKSMA, ed., *ISO 8807, LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[27] ——, *On the Design of Extended LOTOS*, PhD thesis, University of Twente, Twente, 1988.

[28] J. CARLIER, P. CHRETIENNE, AND C. GIRAULT, *Modelling scheduling problems with Timed Petri Nets*, in Advances in Petri Nets 1984, G. Rozenberg, ed., vol. 188 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1984, pp. 62–82.

[29] P.P. CHEN, *The Entity-Relationship Model: Towards a unified view of Data*, ACM Transactions on Database Systems, 1 (1976), pp. 9–36.

[30] G. CHIOLA, C. DUTHEILLET, G. FRANCESCHINIS, AND S. HADDAD, *On well-formed coloured nets and their symbolic reachability graph*, in Proceedings of the 11th International Conference on Applications and Theory of Petri Nets, Paris, June 1990, pp. 387–411.

[31] P. CHRETIENNE, *Les réseaux de petri temporisés*, PhD thesis, Univ. Paris VI, Paris, 1983.

[32] J.M. COLOM AND M. SILVA, *Convex geometry and semiflows in P/T nets, A comparative study of algorithms for computation of minimal P-semiflows*, in Advances in Petri Nets 1990, G. Rozenberg, ed., vol. 483 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1990, pp. 79–112.

[33] O.J. DAHL AND K. NYGAARD, *SIMULA: An ALGOL Based Simulation Language*, Communications of the ACM, 1 (1966), pp. 671–678.

[34] A.M. DAVIS, *Software Requirements: analysis and specification*, Prentice-Hall, Englewood Cliffs, 1990.

[35] A.C.J. DE LEEUW, *Systeemleer en Organisatiekunde* (in Dutch), Stenfert Kroese, Leiden, 1974.

[36] J. DEPREE AND C. SWARTZ, *Introduction to Real Analysis*, John Wiley and Sons, New York, 1988.

[37] E.W. DIJKSTRA, *A note on two problems in connection with graphs*, Numerische Mathematik, 1 (1959), pp. 269–271.

[38] C. Dutheillet and S. Haddad, *Regular Stochastic Petri Nets*, in Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, Bonn, June 1989.

[39] S.E. Elmaghraby, *The role of modelling in I.E. design*, Journal of Industrial Engineering, 14 (1968).

[40] Y. Ermoliev, S. Uryas'ev, and J. Wessels, *On the optimization of material flow systems via simulation*, IIASA-Working Paper, International Institute for Applied Systems Analysis, Laxenburg, (1992).

[41] G. Florin and S. Natkin, *Evaluation based upon Stochastic Petri Nets of the Maximum Throughput of a Full Duplex Protocol*, in Application and theory of Petri nets : selected papers from the first and the second European workshop, C. Girault and W. Reisig, eds., vol. 52 of Informatik Fachberichte, Berlin, 1982, Springer-Verlag, New York, pp. 280–288.

[42] D.W. Fogarty and T.R. Hoffmann, *Production and inventory management*, South-Western,Cincinnati, 1983.

[43] H.J. Genrich, *Projection of CE-Systems*, in Advances in Petri Nets 1985, G. Rozenberg, ed., vol. 222 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1985, pp. 224–232.

[44] ——, *Predicate/Transition-Nets*, in Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties, W. Brauer, W. Reisig, and G. Rozenberg, eds., vol. 254 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1987, pp. 207–247.

[45] H.J. Genrich and K. Lautenbach, *The Analysis of Distributed Systems by means of Predicate/Transition-Nets*, in Semantics of Concurrent Compilation, G. Kahn, ed., vol. 70 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1979, pp. 123–146.

[46] ——, *System modelling with high level Petri nets*, Theoretical Computer Science, 13 (1981), pp. 109–136.

[47] R. Gerber and I. Lee, *A calculus for communicating shared resources*, in Proceedings of CONCUR 1990, J.C.M. Baeten and J.W. Klop, eds., vol. 458 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1990, pp. 123–146.

[48] D. Harel, *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, 8 (1987), pp. 231–274.

[49] K.M. van Hee and P.M.P. Rambags, *Discrete Event Systems: Dynamic versus Static Topology.* Computing Science Notes 89/09, Eindhoven University of Technology, Eindhoven, 1989.

[50] K.M. van Hee and L.J.A.M. Somers, *System Engineering: a Formal Approach* (to appear), 1992.

[51] K.M. van Hee, L.J. Somers, and M. Voorhoeve, *EXSPECT, the functional part.* Computing Science Notes 88/20, Eindhoven University of Technology, Eindhoven, 1988.

[52] ——, *A Formal Framework for Simulation of Discrete Event Systems*, in Proceedings of the 3rd European Simulation Congress, D. Murray-Smith, J. Stephenson, and R.N. Zobel, eds., Edinburgh, Scotland, September 1989, Simulation Councils Inc., pp. 113–116.

[53] ——, *Executable specifications for distributed information systems*, in Proceedings of the IFIP TC 8 / WG 8.1 Working Conference on Information System Concepts: An In-depth Analysis, E.D. Falkenberg and P. Lindgreen, eds., Namur, Belgium, 1989, Elsevier Science Publishers, Amsterdam, pp. 139–156.

[54] ——, *A Formal Framework for Dynamic Modelling of Information Systems*, in Proceedings of the Workshop on the Next Generation of CASE-tools, S. Brinkemper and G. Wijers, eds., SERC, 1990, pp. E2.1–E2.7.

[55] ——, *A Formal Framework for Dynamic Modelling of Information Systems*, in Dynamic Modelling of Information Systems, H.G. Sol and K.M. van Hee, eds., Elsevier Science Publishers, Amsterdam, 1991, pp. 227–236.

[56] ——, *A Modelling Environment for Decision Support Systems*, Decision Support Systems, 7 (1991), pp. 241–251.

[57] ——, *Z and high level petri nets*, in VDM91 Formal Software Development Methods, S. Prehn and W.J. Toetenel, eds., vol. 551 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1991, pp. 204–219.

[58] K.M. van Hee and P.A.C. Verkoulen, *Integration of a Data Model and Petri Nets*, in Proceedings of the 12th International Conference on Applications and Theory of Petri Nets, Aarhus, June 1991, pp. 410–431.

[59] M. Hennessy, *Algebraic Theory of Processes*, The MIT Press, Cambridge, 1988.

[60] W.H. Hesselink, *Deadlock and Fairness in Morphisms of Transition Systems*, Theoretical Computer Science, 59 (1988), pp. 235–257.

[61] A.A.P. van den Heuvel, *IAT: een tool voor het analyseren van Interval Timed Petri Nets* (in Dutch), Master's thesis, Eindhoven University of Technology, Eindhoven, 1991.

[62] H.P. Hillion and J.P Proth, *Performance Evaluation of Job-Shop Systems Using Timed Event Graphs*, IEEE Transactions on Automatic Control, 34 (1989), pp. 3–9.

[63] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, 1985.

[64] M.A. Holliday and M.K. Vernon, *A Generalised Timed Petri Net Model for Performance Analysis*, IEEE Transactions on Software Engineering, 13 (1987), pp. 1279–1310.

[65] A.W. Holt, H. Saint, R. Shapiro, and S. Warshall, *Final Report on the Information Systems Theory Project*, Tech. Rep. RADC-TR-68-305, Griffiss Air Force Base, New York, 1968.

[66] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Cambridge, 1979.

[67] P. Hubner, A.M. Jensen, L.O. Jepsen, and K. Jensen, *Reachability trees for high level Petri nets*, Theoretical Computer Science, 45 (1986), pp. 261–292.

[68] N.E. Hutchinson, *An integrated approach to logistics management*, Prentice-Hall, Englewood Cliffs, 1987.

[69] K. Jensen, *Coloured Petri Nets and the invariant-method*, Theoretical Computer Science, 14 (1981), pp. 317–336.

[70] ——, *Coloured Petri Nets*, in Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties, W. Brauer, W. Reisig, and G. Rozenberg, eds., vol. 254 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1987, pp. 248–299.

[71] ——, *Coloured Petri Nets: A High Level Language for System Design and Analysis*, in Advances in Petri Nets 1990, G. Rozenberg, ed., vol. 483 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1990, pp. 342–416.

[72] K. Jensen and G. Rozenberg, eds., *High-level Petri Nets: Theory and Application*, Springer-Verlag, New York, 1991.

[73] C.B. Jones, *Systematic Development using VDM*, International Series in Computer Science, Prentice-Hall, Englewood Cliffs, 1986.

[74] N.D. Jones, L.H. Landweber, and Y.E Lien, *Complexity of some problems in Petri nets*, Theoretical Computer Science, 4 (1977), pp. 277–299.

[75] R. Koymans, *Specifying real-time properties with metric temporal logic*, Real-Time Systems, 2 (1990), pp. 255–299.

[76] C. Lin and D.C. Marinescu, *On Stochastic High-Level Petri Nets*, in Proceedings of the International Workshop on Petri Nets and Performance Models, IEEE Computer Society Press, Madison, 1987, pp. 34–43.

[77] D. Lock, *Project Management Handbook*, Gower Technical Press, Aldershot, 1987.

[78] M. Lundeberg, G. Goldkuhl, and A. Nilsson, *Information systems development : a systematic approach*, tech. rep., publ. of the Royal Institute of Technology (Stockholm) and University of Stockholm, 1978.

[79] D.A. Marca and C.L. McGowan, *SADT : structured analysis and design technique*, McGraw-Hill, New York, 1988.

[80] M. Ajmone Marsan, *Stochastic Petri Nets: An Elementary Introduction*, in Advances in Petri Nets 1989, G. Rozenberg, ed., vol. 424 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1990, pp. 1–29.

[81] M. Ajmone Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani, *On Petri Nets with Stochastic Timing*, in Proceedings of the International Workshop on Timed Petri Nets, Torino, 1985, IEEE Computer Society Press, pp. 80–87.

[82] M. Ajmone Marsan, G. Balbo, and G. Conte, *A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems*, ACM Transactions on Computer Systems, 2 (1984), pp. 93–122.

[83] ——, *Performance Models of Multiprocessor Systems*, The MIT Press, Cambridge, 1986.

[84] J. Martinez and M. Silva, *A simple and fast algorithm to obtain all invariants of a generalised Petri Net*, in Application and theory of Petri nets : selected papers from the first and the second European workshop, C. Girault and W. Reisig, eds., vol. 52 of Informatik Fachberichte, Berlin, 1982, Springer-Verlag, New York, pp. 301–310.

[85] S. Mauw, *Process algebra as a tool for the specification and verification of CIM-architectures*, in Applications of process algebra, J.C.M. Baeten, ed., vol. 17 of Cambridge Tracts in TCS, Cambridge University Press, 1990, pp. 53–81.

[86] S. Mauw and G.J. Veltink, *A Process Specification Formalism*, in Fundamenta Informaticae XIII, 1990, pp. 82–139.

[87] H.C. Meal, *Putting production decisions where they belong*, Harvard Business review, 84 (1984), pp. 102–111.

[88] G. Memmi and G. Roucairol, *Linear algebra in net theory*, in Net theory and applications : Proceedings of the advanced course on general net theory,processes and systems (Hamburg, 1979), W. Brauer, ed., vol. 84 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1980, pp. 213–223.

[89] P. Merlin, *A Study of the Recoverability of Computer Systems*, PhD thesis, University of California, Irvine, California, 1974.

[90] P. Merlin and D.J. Faber, *Recoverability of communication protocols*, IEEE Transactions on Communication, 24 (1976), pp. 1036–1043.

[91] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1980.

[92] M.K. Molloy, *On the Integration of Delay and Throughput Measures in Distributed Processing Models*, PhD thesis, University of California, Los Angeles, 1981.

[93] T. Murata, *Petri Nets: Properties, Analysis and Applications*, Proceedings of the IEEE, 77 (1989), pp. 541–580.

[94] M. Odijk, *ITPN analysis of ExSpect specifications with respect to production logistics*, Master's thesis, Eindhoven University of Technology, Eindhoven, 1991.

[95] J.S. Ostroff, *Temporal Logic for Real-Time Systems*, John Wiley and Sons, New York, 1989.

[96] ——, *Survey of Formal Methods for the Specification and Design of Real-Time Systems*, Journal of Systems and Software, 18 (April 1992).

[97] A. Pagnoni, *Stochastic Nets and Performance Evaluation*, in Advances in Petri Nets 1986 Part I: Petri Nets, central models and their properties, W. Brauer, W. Reisig, and G. Rozenberg, eds., vol. 254 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1987, pp. 460–478.

[98] ——, *Project Engineering, Computer-Oriented Planning and Operational Decision Making*, Springer-Verlag, New York, 1990.

[99] J.L. Peterson, *A Note on Colored Petri Nets*, Information Processing Letters, 11 (1980), pp. 40–43.

[100] ——, *Petri net theory and the modeling of systems*, Prentice-Hall, Englewood Cliffs, 1981.

[101] C.A. Petri, *Kommunikation mit Automaten*, PhD thesis, Institut für instrumentelle Mathematik, Bonn, 1962.

[102] ——, *Introduction to general net theory*, in Net theory and applications : Proceedings of the advanced course on general net theory,processes and systems (Hamburg, 1979), W. Brauer, ed., vol. 84 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1980, pp. 1–20.

[103] M. Pidd, *Computer modelling for discrete simulation*, John Wiley and Sons, New York, 1989.

[104] A. Pnueli, *The temporal logic of programs*, in Proceedings of the 18th IEEE Annual Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Providence, 1977, pp. 46–57.

[105] W.L. Price, *Graphs and networks, an introduction*, Butterworths, London, 1971.

[106] S. Raczynski, *Graphical description and a program generator for queueing models*, Simulation, 55 (1990), pp. 147–152.

[107] C.V. Ramamoorthy and G.S. Ho, *Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets*, IEEE Transactions on Software engineering, 6 (1980), pp. 440–449.

[108] C. Ramchandani, *Performance Evaluation of Asynchronous Concurrent Systems by Timed Petri Nets*, PhD thesis, Massachusetts Institute of Technology, Cambridge, 1973.

[109] R.R Razouk and C.V. Phelps, *Performance analysis using Timed Petri Nets*, in Proceedings of the 1984 International Conference on Parallel Processing, IEEE Computer Society Press, Ohio, 1984, pp. 126–128.

[110] G.M. Reed and A.W. Roscoe, *A timed model for communicating sequential processes*, Theoretical Computer Science, 58 (1988), pp. 249–261.

[111] W. Reisig, *Petri nets: an introduction*, Prentice-Hall, Englewood Cliffs, 1985.

[112] R.E. Shannon, *Systems simulation: the art and science*, Prentice-Hall, Englewood Cliffs, 1975.

[113] J. Sifakis, *Use of Petri Nets for performance evaluation*, in Proceedings of the Third International Symposium IFIP W.G. 7.3., Measuring, modelling and evaluating computer systems (Bonn-Bad Godesberg, 1977), H. Beilner and E. Gelenbe, eds., Elsevier Science Publishers, Amsterdam, 1977, pp. 75–93.

[114] ——, *Performance Evaluation of Systems using Nets*, in Net theory and applications : Proceedings of the advanced course on general net theory, processes and systems (Hamburg, 1979), W. Brauer, ed., vol. 84 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1980, pp. 307–319.

[115] M. Silva and R. Valette, *Petri Nets and Flexible Manufacturing*, in Advances in Petri Nets 1989, G. Rozenberg, ed., vol. 424 of Lecture Notes in Computer Science, Springer-Verlag, New York, 1990, pp. 274–417.

[116] J.M. Spivey, *The Z Notation: A Reference Manual*, Prentice-Hall, Englewood Cliffs, 1989.

[117] Y. Sugimori, K. Kusunoki, F. Cho, and S. Uchikawa, *Toyota Production System and Kanban System, Materialisation of Just-in-time and Respect-for-human System* , International Journal of Production Research, 15 (1977), pp. 553–564.

[118] R. Suri, *Perturbation analysis: the state of the art and research issues explained via the GI/G/1 queue*, Proceedings of the IEEE, 77 (1989), pp. 114–137.

[119] J.D. Ullman, *Principles of database and knowledge-base systems*, Computer Science Press, Rockville, 1988.

[120] A. Valmari, *Stubborn sets for reduced state space generation*, in Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, Bonn, June 1989.

[121] P.T Ward and S.J. Mellor, *Structured development for real-time systems*, Yourdon, London, 1985.

[122] J. Wessels, *Tools for the Interfacing between Dynamical Problems within Decision Support Systems.* COSOR-memorandum 91-29, Eindhoven University of Technology, Eindhoven, 1991.

[123] ——, *Decision systems; the relation between problem specification and mathematical analysis*, in User-oriented decision support (to appear), J. Wessels and A.P. Wierzbicki, eds., 1992.

[124] J.C. Wetherbe, *Systems analysis for computer based information systems*, West Publishing Company, New York, 1979.

[125] G.E. Whitehouse, *Systems analysis and design using network techniques*, Prentice-Hall, Englewood Cliffs, 1973.

[126] W. Whitt, *The Queueing Network Analyser*, The BELL Systems Technical Journal, 62 (1983).

[127] H.S. Wilf, *Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, 1986.

[128] C.Y. Wong, T.S. Dillon, and K.E. Forward, *Timed Places Petri Nets with Stochastic Representation of Place Time*, in Proceedings of the International Workshop on Timed Petri Nets, Torino, 1985, IEEE Computer Society Press, pp. 96–103.

[129] D. Wood, *Theory of Computation*, Harper and Row, New York, 1987.

[130] E. Yourdon, *Managing the system life cycle*, Yourdon, London, 1982.

[131] A. Zenie, *Coloured Stochastic Petri Nets*, in Proceedings of the International Workshop on Timed Petri Nets, Torino, 1985, IEEE Computer Society Press, pp. 262–271.

[132] C.R. Zervos, *Coloured Petri Nets: their properties and applications*, PhD thesis, University of Michigan, Michigan, 1977.

[133] W.M. Zuberek, *Timed Petri Nets and Preliminary Performance Evaluation*, in Proceedings of the 7th annual Symposium on Computer Architecture, vol. 8(3) of Quarterly Publication of ACM Special Interest Group on Computer Architecture, 1980, pp. 62–82.

Samenvatting

Het in dit proefschrift beschreven onderzoek richt zich op het modelleren en analyseren van complexe dynamische systemen. Dit onderzoek heeft, onder andere, geresulteerd in een aantal concepten en technieken, welke algemeen bruikbaar zijn in situaties waar de voortgang bepaald wordt door discrete gebeurtenissen. Ondanks het feit dat een belangrijk deel van de resultaten algemeen toepasbaar is, ligt in dit proefschrift de nadruk vooral op toepassingen in de logistiek.

De beschreven aanpak is gebaseerd op een Petri net model, uitgebreid met 'tijd' en 'kleur'. Dit Petri net model is uitermate geschikt voor het modelleren van logistieke systemen. Immers, met dit model is het mogelijk de logistieke stromen (goederen, middelen en informatie) op een natuurlijke en eenvormige wijze te beschrijven. Ook is het mogelijk om het gedistribueerde aspect van een logistiek systeem op een inzichtelijke wijze te representeren.

Het doel van het in dit proefschrift beschreven onderzoek valt uiteen in twee delen. Enerzijds moet het proefschrift gereedschappen leveren ter ondersteuning van het modelleren van discrete dynamische systemen, in het bijzonder logistieke systemen. Anderzijds is het de bedoeling een bijdrage leveren aan de ontwikkeling van bruikbare methoden voor de analyse van Petri nets.

In hoofdstuk 2 wordt het *Interval Timed Coloured Petri Net* (ITCPN) model geïntroduceerd. Dit model dient als uitgangspunt voor de rest van het proefschrift. Het ITCPN-model wijkt af van reeds bestaande Petri net modellen doordat tokens een tijdstempel dragen en doordat tijdsduren beschreven worden door middel van een interval, d.w.z. een onder- en bovengrens. In dit hoofdstuk formuleren we ook de vragen die we graag beantwoord willen zien.
Hoofdstuk 3 richt zich op de analyse van Petri nets uitgebreid met 'tijd' en 'kleur'. Er worden drie methoden behandeld waarmee ITCPN's geanalyseerd kunnen worden. Eén van deze methoden, de MTSRT methode, kan gebruikt worden voor de analyse van een willekeurig ITCPN, terwijl de andere twee methoden alleen toegepast kunnen worden op een beperkte, doch zinvolle, klasse ITCPN's.
Ter ondersteuning van het werken met ITCPN's is er ook software ontwikkeld. Deze

software maakt deel uit van het pakket ExSpect dat binnen de vakgroep Informatica van de Technische Universiteit Eindhoven is ontwikkeld. ExSpect maakt gebruik van een specificatietaal welke gebaseerd is op een Petri net model dat veel overeenkomsten vertoont met het in hoofdstuk 2 geïntroduceerde ITCPN-model. In hoofdstuk 4 behandelen we enkele aspecten van deze taal en beschrijven we de onderliggende software. Met name besteden we aandacht aan het ontwerp- en analyse-gereedschap van ExSpect, welke voor een belangrijk deel door de auteur van dit proefschrift ontwikkeld zijn.

In hoofdstuk 5 beschrijven we hoe we een logistiek systeem op een gestructureerde wijze kunnen modelleren. Dit doen we door een systematische indeling te geven van de logistieke stromen en processen. Deze indeling is als uitgangspunt gebruikt voor de ontwikkeling van een bibliotheek bestaande uit logistieke componenten. Deze componenten zijn in ExSpect gespecificeerde (sub)systemen. Op deze wijze is het mogelijk om in korte tijd een reëel logistiek systeem op een inzichtelijke wijze te modelleren. In zekere zin vormt dit hoofdstuk een eerste aanzet voor een 'referentiemodel' voor de logistiek.

Curriculum vitae

De schrijver van dit proefschrift werd op 29 januari 1966 geboren te Eersel. Van 1978 tot 1984 bezocht hij het Rythoviuscollege aldaar.
Na het behalen van het VWO-diploma, begon hij in 1984 zijn studie aan de Technische Universiteit Eindhoven in de richting Informatica. Deze studie werd in september 1988 afgesloten middels een doctoraalscriptie over het gebruik van ExSpect als simulatietaal. Het hiervoor benodigde onderzoek werd uitgevoerd onder toezicht van prof.dr. K.M. van Hee.
Sinds oktober 1988 is de schrijver als toegevoegd onderzoeker verbonden aan de vakgroep Besliskunde en Stochastiek van de Technische Universiteit Eindhoven. Het betreft een promotieplaats in het kader van het TASTE-project. Dit proefschrift is het resultaat van het onderzoek dat de schrijver in de afgelopen vier jaar heeft verricht onder de begeleiding van prof.dr. J. Wessels en prof.dr. K.M. van Hee.