Petri net based scheduling

W.M.P. van der Aalst

Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB, Eindhoven, The Netherlands, telephone: -31 40 2474295, fax: -31 40 2463992, e-mail: wsinwa@win.tue.nl

Abstract.

Timed Petri nets can be used to model and analyse scheduling problems. To support the modelling of scheduling problems, we provide a method to map tasks, resources and constraints onto a timed Petri net. By mapping scheduling problems onto Petri nets, we are able to use standard Petri net theory. In this paper we will show that we can use Petri net based tools and techniques to find conflicting and redundant precedences, upper- and lowerbounds for the makespan, etc. This is illustrated by a Petri net based analysis of the notorious 10×10 problem due to Fisher & Thompson (1963).

Keywords: Scheduling, Timed Petri nets, Analysis of Petri nets.

1 Introduction

During the last two decades, much research has been done simultaneously on Petri nets and scheduling problems. The results achieved in both research areas have been applied to production systems, logistic systems and computer systems. Although scheduling techniques and Petri nets focus on the same application domains, there has been little effort to put these activities in gear with each other. The aim of this paper is to provide a link between Petri nets and scheduling problems.

The optimal allocation of scarce resources to tasks over time has been the prime subject of research on scheduling problems. Despite the inherent complexity of many scheduling problems, effective algorithms have been developed. However, most researchers focussed on the effectiveness of the algorithms, discarding the issue of flexibility.
Research on Petri nets addresses the issue of flexibility; many extensions have been proposed to facilitate the modelling of complex systems. Typical extensions are the addition of 'colour', 'time' and 'hierarchy' (Jensen, 1992; Jensen & Rozenberg, 1991; Aalst, 1992a). Petri nets extended with these features are suitable for the representation and study of the complex industrial systems of the 90's. These Petri nets inherit all the advantages of the classical Petri net, such as the graphical and precise nature, the firm mathematical foundation and the abundance of analysis methods. Moreover, adequate computer tools have been put on the market. These tools support both the modelling and analysis of Petri nets extended with 'colour', 'time' and 'hierarchy'. Therefore, it is interesting

to investigate the application of Petri nets to scheduling. In this paper we concentrate on timed Petri nets, i.e. Petri nets extended with a timing concept.

As we will show, it is relatively easy to map scheduling problems onto timed Petri nets. However, the application of Petri net based analysis techniques to a scheduling problem represented by a timed Petri net is far from trivial! Therefore, we report 7 useful results. Each of these results shows how a specific aspect of a scheduling problem can be analysed by applying a standard Petri net based analysis technique. For example, we will show that we can find conflicting and redundant precedences, and upper- and lowerbounds for the makespan of a scheduling problem.

Using Petri nets for the modelling of scheduling problems is not a new idea. However, most of the results in this area focus on cyclic scheduling problems. Ramamoorthy & Ho (1980) and Hillion & Proth (1989) have used a technique based on a 'marked graphs' (a subclass of Petri nets) to analyse the throughput of cyclic (production) processes. Carlier, Chretienne & Girault (1984, 1988, 1983), Gao, Wong & Ning (1991) and Watanabe & Yamauchi (1993) also focussed on minimal cycle times for repetitive scheduling problems. In this paper we focus on the traditional non-cyclic scheduling problems such as machine scheduling and job-shop scheduling (Pinedo, 1995). We also use a different timed Petri net model. In our model time is associated with tokens instead of transitions.

The remainder of this paper is organized as follows. Section 2 introduces a *timed Petri net* model. This Petri net model has been extended with a timing concept. Section 3 describes the type of scheduling problems we are going to address. Section 4 is devoted to the mapping of scheduling problems onto timed Petri nets. The usefulness of the Petri net analysis techniques in the context of scheduling is discussed in Section 5. Section 6 discusses the flexibility of the approach presented in this paper. Finally, this approach is applied to the notorious 10×10 problem due to Fisher & Thompson (1963).

2 Timed Petri nets

Historically speaking, Petri nets originate from the early work of Carl Adam Petri (1962). Since then the use and study of Petri nets has increased considerably. For a review of the history of Petri nets and an extensive bibliography the reader is referred to Murata (1989). The classical 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, p is called an *output place* of t if there exists a directed arc from t to p.
We will use the net shown in Figure 1 to illustrate the classical Petri net model. This Petri net models a machine which processes jobs and has two states (free and busy). There are four places (*in*, *free*, *busy* and *out*) and two transitions (*start* and *finish*). In the state shown

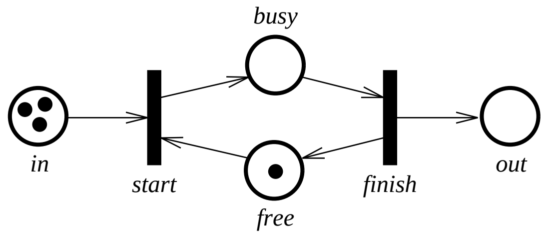

Figure 1: A Petri net which represents a machine.

in Figure 1 there are four tokens; three in place *in* and one in place *free*. The tokens in place *in* represent jobs to be processed by the machine. The token in place *free* indicates that the machine is free and ready to process a job. If the machine is processing a job, then there are no tokens in *free* and there is one token in *busy*. The tokens in place *out* represent jobs which have been processed by the machine. Transition *start* has two input places (*in* and *free*) and one output place (*busy*). Transition *finish* has one input place (*busy*) and two output places (*out* and *free*).

A transition is called *enabled* if each of its input places contains at least one token. 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'.

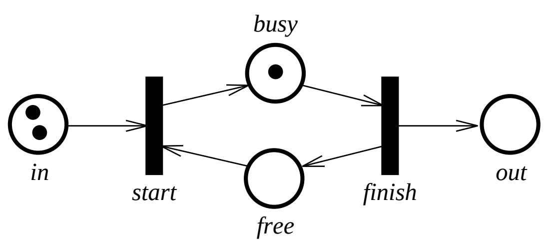

Figure 2: Transition *start* has fired.

Transition *start* is enabled in the state shown in Figure 1, because each of the input places (*in* and *free*) contains a token. Transition *finish* is not enabled because there are no tokens in place *busy*. Therefore, transition *start* is the only transition that can fire. Firing transition *start* means consuming two tokens, one from *in* and one from *free*, and producing one token for *busy*. The resulting state is shown in Figure 2. In this state only transition *finish* is enabled. Hence, transition *finish* fires and the token in place *busy* is consumed and two tokens are produced, one for *out* and one for *free*. Now transition *start* is enabled, etc. Note that as long as there are jobs waiting to be processed, the two transitions fire alternately, i.e. the machine modelled by this net can only process one job at a time.

Adding time

For real systems it is often important to describe the *temporal behaviour* of the system, i.e. we need to model durations and delays. Since the classical Petri net is not easily capable of handling quantitative time, we add a timing concept. There are many ways to introduce time into the classical Petri net (Aalst, 1992b). In this paper a timing mechanism is used where time is associated with tokens, and transitions determine delays (Aalst, 1993).

Each token has a *timestamp* which models the time the token becomes available for consumption. Since these timestamps indicate when tokens become available, a transition becomes enabled the earliest moment for which each of its input places contains a token which is available. The timestamp of a produced token is equal to the *firing time* plus the *firing delay* of the corresponding transition. Consider the net shown in Figure 1. If place *in* contains one token with timestamp 1 and place *free* contains a token with timestamp 0, then transition *start* becomes enabled at time 1. If the firing delay of *start* is equal to 3, then the produced token for place *busy* has timestamp 1+3=4.
Firing is atomic, i.e. the moment a transition consumes tokens from the input places the produced tokens appear in the output places. However, because of the firing delay it takes some time before the produced tokens become available for consumption.
This results in the following definition of a *timed Petri net*.

Definition 1
A *timed Petri net* is a six tuple TPN $= (P, T, I, O, TS, D)$ satisfying the following requirements:

(i) P is a finite set of *places*.

(ii) T is a finite set of *transitions*.

(iii) $I \in T \rightarrow \mathcal{P}(P)$ is a function which defines the set of *input places* of each transition.

(iv) $O \in T \rightarrow \mathcal{P}(P)$ is a function which defines the set of *output places* of each transition.

(v) TS is the *time set*.

(vi) $D \in T \rightarrow TS$ is a function which defines the *firing delay* of each transition.

The *state* of a timed Petri net is given by the distribution of tokens over the places and the corresponding timestamps. Firing a transition results in a new state. This way we can generate a sequence of states $s_0, s_1, ...s_n$ such that s_0 is the initial state and s_{i+1} is the state reachable from s_i by firing a transition. Transitions are *eager*, i.e. they fire as soon as possible. If several transitions are enabled at the same time, then any of these transitions may be the next to fire. Therefore, in general, many *firing sequences* are possible. Let s_0 be the initial state of a timed Petri net. A state is called a *reachable state* if and only if there exists a firing sequence $s_0, s_1, ...s_n$ which 'visits' this state. A *terminal state* is a state where none of the transitions is enabled, i.e. a state without successors.

We will use an example to illustrate Definition 1. Figure 3 shows a Petri net which models two identical parallel machines. The tokens in place *in* represent jobs which need to be processed by one of the two machines. The timestamps of these tokens are shown in Figure 3. The first job arrives at time 0, the second at time 1 and the third at time 3. The token in place *free1* has timestamp 0. This means that at time 0 the first machine is free

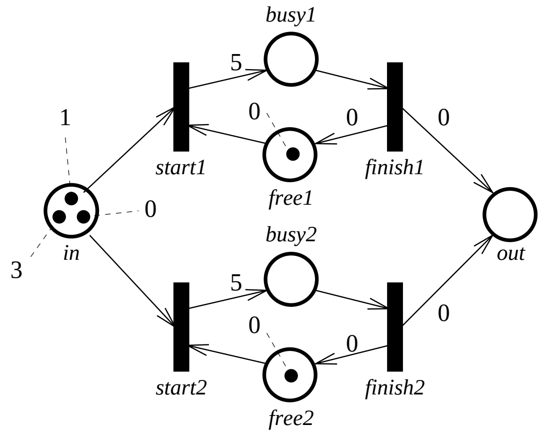

Figure 3: Two parallel identical machines.

and ready to process a job. At time 0, the other machine is also free and ready to process a job. Therefore, one of the two machines will start processing the job that arrives at time 0. Let us assume that the first machine takes care of this job, i.e. transition *start1* fires at time 0 and produces a token with a delay equal to 5 time units (e.g. minutes). As a result *free1* is empty and *busy1* contains a token with timestamp 5. At time 1 transition *start2* fires, i.e. the second machines starts to process the job arriving at time 1. The resulting state is shown in Figure 4. (Note that place *busy2* contains a token with timestamp $1 + 5 = 6$.)

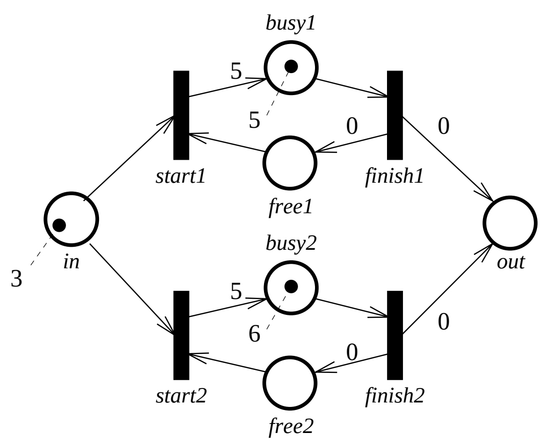

Figure 4: The state resulting from firing *start1* and *start2*.

In this state none of the transitions *start1* and *start2* is enabled since both machines are busy. At time 5 transition *finish1* fires, thus enabling transition *start1*. Transition *start1* also fires at time 5. At time 6 *finish2* fires, followed by the firing of transition *finish1* at time 8. The resulting state is a terminal state. In this terminal state there are 3 tokens in place *out* indicating that the three jobs have been processed successfully. The timestamps of these tokens are equal to the corresponding completion time, i.e. 5, 6 and 8. Note that this is not the only possible firing sequence. If *start2* fires at time 0, alternative states are visited. However, both firing sequences result in the same terminal state.

3 The general scheduling problem

Scheduling is concerned with the optimal allocation of scarce resources to tasks over time (Lawler, Lenstra, Rinnooy Kan & Shmoys, 1993). Scheduling techniques are used to answer questions that arise in production planning, project planning, computer control, manpower planning, etc. Many techniques have been developed for a variety of problem types. To fix the terminology, we begin by defining the 'general scheduling problem'. Many problem types fit into this definition. Moreover, some extensions are discussed in Section 6.

In essence, scheduling boils down to the allocation of *resources* to *tasks* over time. Some authors refer to resources as 'machines' or 'processors' and tasks are also called 'operations' or 'steps of a job'. Resources are used to *process* tasks. However, it is possible that the execution of a task requires more that one resource, i.e. a task is processed by a *resource set*. Moreover, there may be multiple resource sets that are capable of processing a specific task. The *processing time* of a task is the time required to execute the task given a specific resource set. By adding *precedence constraints* it is possible to formulate requirements about the order in which the tasks have to be processed. We will assume that resources are always available, but we shall not necessarily assume the same for tasks. Each task has a *release time*, i.e. the time at which the task becomes available for processing. This leads to the following definition.

Definition 2
A *scheduling problem* is a 6-tuple SP $= (T, R, PRE, TS, RT, PT)$ satisfying the following requirements.

(i) T is a finite set of *tasks*.

(ii) R is a finite set of *resources*.

(iii) $PRE \subseteq T \times T$ is a partial order, the *precedence relation*.

(iv) TS is the *time set*.

(v) $RT \in T \to TS$ is a function which defines the *release time* of each task.

(vi) $PT \in (T \times \mathcal{P}(R)) \not\to TS$ defines for each task t:[1]

  (a) the resource sets *capable* of processing task t and

  (b) the *processing time* required to process t by a specific resource set.

This definition specifies the data required to formulate a scheduling problem. The tasks are denoted by T and the resources are denoted by R. The precedence relation PRE is used to specify precedence constraints. If task t has to be processed before task t', then $\langle t, t' \rangle \in PRE$, i.e. the execution of task t has to be completed before the execution of task t' may start. TS is the *time set*. $\mathbb{N}$ and $\mathbb{R}^+ \cup \{0\}$ are typical choices for TS. The

[1] $A \not\to B$ denotes the set of all partial functions from A to B. $\mathcal{P}(A)$ is the powerset of A.

release time $RT(t)$ of a task t specifies the time at which the task becomes available for processing, i.e. the execution of t may not start before time $RT(t)$. Function PT specifies two things; (1) the resource sets capable of processing task t:

$$\{rs \in \mathcal{P}(R) \mid \langle t, rs\rangle \in dom(PT)\}$$

and (2) the processing time required to process t by a specific resource set rs:

$$PT(\langle t, rs\rangle)$$

To clarify this definition we present a small example.

Example: a job-shop

Consider a job-shop where two jobs $\{J1, J2\}$ have to be processed by two machines $\{M1, M2\}$. Job $J1$ requires two operations A and B. Operation A is processed by machine $M1$ followed by operation B processed by machine $M2$. The processing time of both operations is equal to 3 minutes. Job $J2$ requires only one operation C. This operation can be processed by one machine $M1$ or both machines at the same time (i.e. $M1$ and $M2$). Processing operation C by machine $M1$ takes 5 minutes, processing operation C by machine $M1$ and machine $M2$ takes only 2 minutes. Both jobs $J1$ and $J2$ enter the jobshop at time zero.

The corresponding scheduling problem SP $= (T, R, PRE, TS, RT, PT)$ is specified as follows:

$$\begin{aligned}
T &= \{J1_A, J1_B, J2_C\} \\
R &= \{M1, M2\} \\
PRE &= \{\langle J1_A, J1_B\rangle\} \\
TS &= \mathbb{R}^+ \cup \{0\} \\
RT(J1_A) &= RT(J1_B) = RT(J2_C) = 0 \\
dom(PT) &= \{\langle J1_A, \{M1\}\rangle, \langle J1_B, \{M2\}\rangle, \langle J2_C, \{M1\}\rangle, \\
&\quad \langle J2_C, \{M1, M2\}\rangle\} \\
PT(\langle J1_A, \{M1\}\rangle) &= 3 \\
PT(\langle J1_B, \{M2\}\rangle) &= 3 \\
PT(\langle J2_C, \{M1\}\rangle) &= 5 \\
PT(\langle J2_C, \{M1, M2\}\rangle) &= 2
\end{aligned}$$

Each operation corresponds to a task. The precedence relation specifies the constraint that the operations in job $J1$ have to be processed in a specific order. The domain of function PT signifies the resource sets able to process a specific operation. The processing times are also specified by PT.

Assumptions

Although Definition 2 is quite general we have made a number of assumptions about the structure of a scheduling problem:

1. No resource may process more than one task at a time.
2. Each resource is continuously available for processing.
3. No pre-emption, i.e. each operation, once started, must be completed without interruptions.
4. The processing times are independent of the schedule. Moreover, the processing times are fixed and known in advance.

A *schedule* is an allocation of resources to tasks over time. Such a schedule can be represented by a function $s \in T \rightarrow (\mathcal{P}(R) \times TS)$, i.e. for each task it is specified *when* it is processed by *which* resources. If t is a task and $\langle rs, st \rangle \in s(t)$, then t is processed by resource set rs starting at time st. Note that given a schedule and a task t, we can define (1) $st(t)$: the start time of t, (2) $ct(t)$: the completion time of t and (3) $ra(t)$: the resource set that is used to process t.
A schedule $s \in T \rightarrow (\mathcal{P}(R) \times TS)$ is *feasible* if the following constraints are satisfied:

1. precedences are obeyed: $\forall_{\langle t,t' \rangle \in PRE}\ ct(t) \leq st(t')$
2. release times are obeyed: $\forall_{t \in T}\ st(t) \geq RT(t)$
3. valid resource sets are used: $\forall_{t \in T}\ \langle t, ra(t) \rangle \in dom(PT)$
4. a resource cannot be used to process multiple tasks at the same time: $\forall_{t,t' \in T}\ (ra(t) \cap ra(t') \neq \emptyset) \Rightarrow (ct(t) \leq st(t')\ \vee\ ct(t') \leq st(t))$

Consider the job-shop example: schedule $s_j = \{\langle J1_A, \langle \{M1\}, 0 \rangle \rangle, \langle J1_B, \langle \{M2\}, 3 \rangle \rangle, \langle J2_C, \langle \{M1\}, 3 \rangle \rangle\}$ is a feasible schedule. In the remainder we will only consider feasible schedules.

Performance measures
There are numerous objectives in scheduling. Therefore, there are dozens of sensible performance measures. In this paper only a few of them are discussed. For summary of these measures, the reader is referred to French (1982) and Baker (1974).
First we define an additional concept: the *due-date* of a task. The due-date d_t of a task t, is the desired completion time of t.
The flow-time of a task t (F_t) is the time between the release of t and the completion of t, i.e. $F_t = ct(t) - RT(t)$. The lateness L_t of a task t is defined as follows: $L_t = ct(t) - d_t$. The tardiness T_t of t only considers 'tardy' tasks, i.e. $T_t = \max(L_t, 0)$. Typical performance measures are the average flow-time of tasks, the average lateness of tasks and the average tardiness of tasks. A related performance measure is the makespan of a set of tasks $M = \max_{t \in T} ct(t)$. In a job-shop the makespan is equal to the total production time. A straightforward objective is to minimize the makespan. Note that for the job-shop example, schedule s_j has a makespan of 8. Since all other feasible schedules have a makespan of at least 8, schedule s_j is *optimal*.
There are many other reasonable objectives, e.g. minimize the number of tardy tasks, minimize the number of waiting tasks, etc.

4 Mapping scheduling problems onto Petri nets

To show that timed Petri nets can be used to model and analyse scheduling problems, we provide a translation from an arbitrary scheduling problem to a ‘suitable’ timed Petri net. This means that we have to map concepts such as tasks, resources and precedences onto places and transitions.

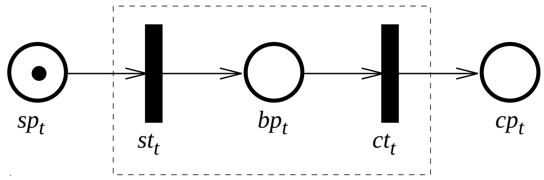

Figure 5: Task t.

Given a task t we identify three stages: (1) t is waiting to be processed, (2) t is being processed and (3) t has been processed. Therefore, we identify two important ‘milestones’: the start time and completion time of t. Basically, Figure 5 shows how we model a task t in terms of a timed Petri net. Transitions st_t and ct_t represent the beginning and termination of t respectively. The places sp_t, bp_t and cp_t correspond to the stages just mentioned. Initially, there is one token in sp_t with timestamp $RT(t)$, the release time of t. Since the token in sp_t becomes available at time $RT(t)$, transition st_t cannot fire before the release time of task t. The firing delay of st_t is equal to the processing time of task t given a specific resource set.

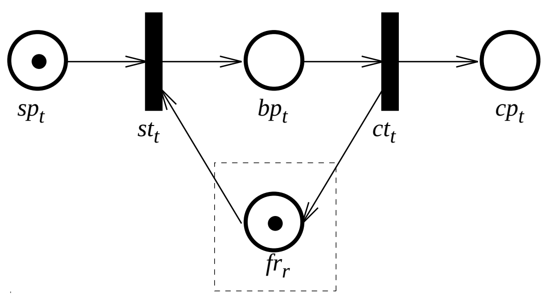

Figure 6: Resource r.

Each *resource* r is modelled by a place fr_r. Initially, fr_r contains one token. Figure 6 shows a resource r which can be used to process a task t. Transition st_t ‘claims’ the resource when the execution of t starts, transition ct_t ‘releases’ the resource when t terminates.

Precedence constraints are modelled by adding extra places. Figure 7 shows the situation where task t precedes task t', i.e. the execution of task t has to be completed before the execution of task t' may start. Place $pre_{\langle t,t' \rangle}$ prevents $st_{t'}$ from firing until ct_t fires. Note that places are used to model the stages of a task, resources and precedences.

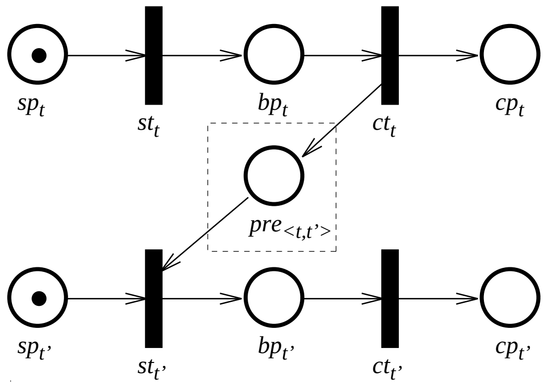

Figure 7: Precedence constraint $\langle t, t' \rangle$.

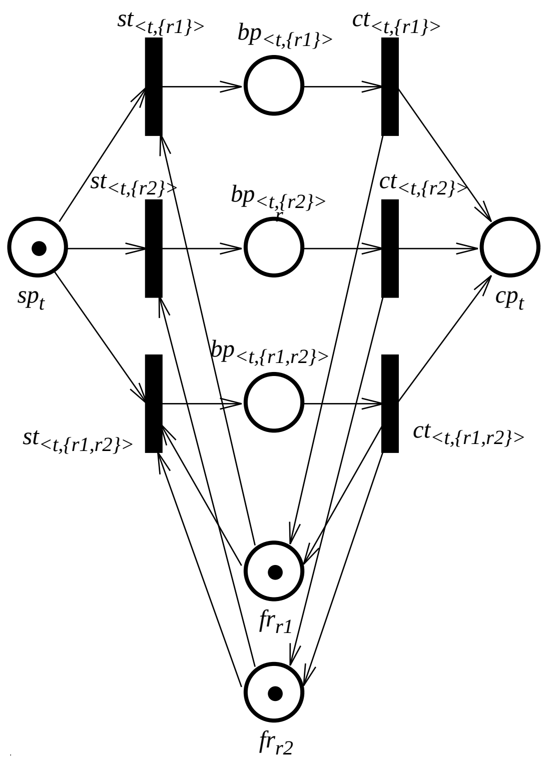

Figure 8: A task with three possible resource sets.

Thus far, we ignored the fact that a task may be processed by one of multiple resource sets. Figure 8 shows how to model this situation. For each resource set rs capable of processing task t, we introduce a place $bp_{\langle t,rs \rangle}$ and two transitions $st_{\langle t,rs \rangle}$ and $ct_{\langle t,rs \rangle}$. Figure 8 shows that task t can be processed by one of the following resource sets: $\{r1\}$, $\{r2\}$ and$\{r1, r2\}$. Note that there is only one 'start place' sp_t and one 'completion place' cp_t.

Consider the job-shop example given in Section 3. Recall that there are three tasks: $J1_A$, $J1_B$ and $J2_C$ and two resources: $M1$ and $M2$. Task $J1_A$ and task $J1_B$ have to be processed by $M1$ and $M2$ respectively. Task $J2_C$ may be processed by $M1$ *or* $M1$ and $M2$. The corresponding Petri net is shown in Figure 9. (The names of places and transitions have been omitted.)

The following definition formalizes the 'recipe' just given.

Definition 3
Given *scheduling problem* SP $= (T, R, PRE, TS, RT, PT)$ we define the corresponding *timed Petri net* TPN $= (\overline{P}, \overline{T}, \overline{I}, \overline{O}, \overline{TS}, \overline{D})$ as follows:

$$\begin{array}{rcl}
\overline{P} & = & \{bp_{\langle t,rs\rangle} \mid \langle t,rs\rangle \in dom(PT)\} \cup \\
 & & \{sp_t \mid t \in T\} \cup \\
 & & \{cp_t \mid t \in T\} \cup \\
 & & \{fr_r \mid r \in R\} \cup \\
 & & \{pre_{\langle t,t'\rangle} \mid \langle t,t'\rangle \in PRE\} \\
\overline{T} & = & \{st_{\langle t,rs\rangle} \mid \langle t,rs\rangle \in dom(PT)\} \cup \\
 & & \{ct_{\langle t,rs\rangle} \mid \langle t,rs\rangle \in dom(PT)\}
\end{array}$$

and for any task $t \in T$ and resource set $rs \in \mathcal{P}(R)$ such that $\langle t,rs\rangle \in dom(PT)$:

$$\begin{array}{rcl}
\overline{I}(st_{\langle t,rs\rangle}) & = & \{sp_t\} \cup \\
 & & \{fr_r \mid r \in R \ \wedge \ r \in rs\} \cup \\
 & & \{pre_{\langle t',t\rangle} \mid t' \in T \ \wedge \ \langle t',t\rangle \in PRE\} \\
\overline{I}(ct_{\langle t,rs\rangle}) & = & \{bp_{\langle t,rs\rangle}\} \\
\overline{O}(st_{\langle t,rs\rangle}) & = & \{bp_{\langle t,rs\rangle}\} \\
\overline{O}(ct_{\langle t,rs\rangle}) & = & \{cp_t\} \cup \\
 & & \{fr_r \mid r \in R \ \wedge \ r \in rs\} \cup \\
 & & \{pre_{\langle t,t'\rangle} \mid t' \in T \ \wedge \ \langle t,t'\rangle \in PRE\} \\
\overline{TS} & = & TS \\
\overline{D}(st_{\langle t,rs\rangle}) & = & PT(\langle t,rs\rangle) \\
\overline{D}(ct_{\langle t,rs\rangle}) & = & 0
\end{array}$$

This definition shows how to model a scheduling problem in terms of a timed Petri net. The initial state of the net is as follows. For each task t, place sp_t contains one token with timestamp $RT(t)$. For each resource r, place fr_r contains one token with timestamp 0. All other places are empty.
We can give an upper bound for the size of the constructed timed Petri net TPN: $\sharp T 2^{\sharp R} + 2\sharp T + \sharp R + (\sharp T)^2$ places and $2\sharp T 2^{\sharp R}$ transitions. However, a typical Petri net representing a scheduling problem with $\sharp T$ tasks and $\sharp R$ resources contains $4\sharp T + \sharp R$ places and $2\sharp T$ transitions.

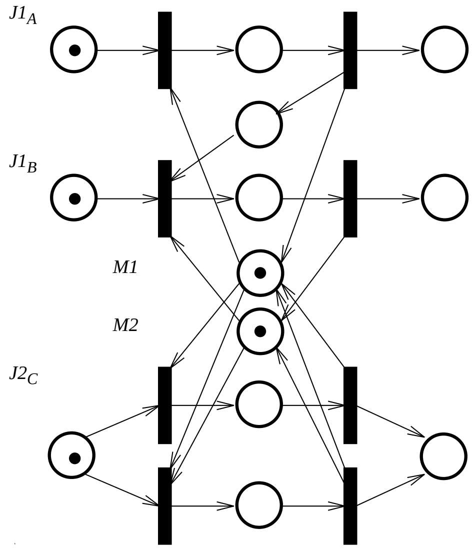

Figure 9: The job-shop scheduling problem.

The mapping of scheduling problems onto Petri nets presented in this section differs from the mappings presented in Carlier, Chretienne & Girault (1984), Gao, Wong & Ning (1991), Hillion & Proth (1989) and Watanabe & Yamauchi (1993). First of all we consider non-cyclic scheduling problems instead of cyclic ones. Secondly, we use a timed Petri net model where time is associated with tokens instead of transitions. Thirdly, we explicitly model the beginning and termination of the processing of a task.

We can use the constructed timed Petri net to calculate feasible schedules. Each firing sequence resulting in a terminal state, represents a possible schedule. Note that these firing sequences have length $2\sharp T+1$. Given such a firing sequence, the resulting terminal state is as follows. For each task t, place cp_t contains one token with timestamp $ct(t)$. For each resource r, place fr_r contains one token with a timestamp equal to the completion time of the last task processed by r. All other places are empty.
It is easy to verify that the schedule represented by such a firing sequence is feasible, i.e.

- precedences are obeyed: the places $pre_{\langle t,t'\rangle}$ take care of this,
- release times are obeyed: transition $st_{\langle t,rs\rangle}$ cannot fire before $RT(t)$,
- valid resources sets are used: transition $st_{\langle t,rs\rangle}$ 'claims' all resources in the resource set rs,
- resources cannot be used to process two tasks at the same time: transition $st_{\langle t,rs\rangle}$ removes all tokens from the places fr_r with $r \in rs$, these tokens are returned the moment task t is completed.

Although each firing sequence of length $2\sharp T + 1$ corresponds to a feasible schedule, the opposite is not true, i.e. there are feasible schedules which do not correspond to any firing sequence. This is a consequence of the fact that transitions are eager to fire, i.e. they

fire as soon as possible. The schedules which correspond to these firing sequences are often referred to as *list schedules*, i.e. schedules where a machine is assigned to a task as soon as possible. Unfortunately, it is not sufficient to consider only list schedules to find an optimal schedule (i.e. a schedule with a minimal makespan). (See for example the introductory example in (French, 1982).) This is the reason most papers about the application of Petri nets to scheduling focus on a subclass of Petri nets: the so-called marked graphs (Carlier, Chretienne & Girault, 1984; Chretienne, 1983; Gao, Wong & Ning, 1991; Hillion & Proth, 1989; Ramamoorthy & Ho, 1980). For this subclass an optimal schedule corresponds to a list schedule.
It is also possible to define an alternative firing-rule for timed Petri nets. This alternative firing-rule relaxes the eagerness-property as follows: a transition that is enabled is allowed to postpone its firing until another transition fires. Note that a postponed transition may become disabled by the firing of another transition. Therefore, we obtain firing sequences which don't correspond to a list schedule. If we adopt this alternative firing-rule, we have *a one-to-one correspondence between the set of feasible schedules and the set of possible firing sequences* (of length $2\sharp T + 1$). Unfortunately, the number of possible firing sequences increases considerably. In the next section we will return to this subject.

5 Analysis

After modelling a scheduling problem in terms of a timed Petri net, an obvious question is "What can we do with the Petri net model?". A major strength of Petri nets is the collection of supporting analysis methods. In this section we discuss the usefulness of these analysis methods in the context of scheduling. First, we will discuss the application of analysis techniques to derive structural properties of the net. Secondly, we will focus on methods to analyse the dynamic behaviour of the constructed timed Petri net.

5.1 Structural properties

Several analysis methods have been developed to find and verify structural properties of classical Petri nets (Martinez & Silva, 1982; Murata, 1989; Reisig, 1985). Here we discuss *place* and *transition invariants*. Place and transition invariants are powerful tools for studying structural properties of Petri nets.
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) of the token. A place invariant holds if the weighted token sum of all tokens remains constant during the execution of the net. Consider for example the net shown in Figure 1. The following two place invariants hold for this net; (1) *free + busy = 1* and (2) *in + busy + out = 3*. The first invariant says that the total number of tokens in the places *free* and *busy* is equal to 1. This means that the machine is either free of busy. The second invariant states that the total number of tokens in the places *in*, *busy* and *out* is equal to 3, the initial number of tokens in *in*. This implies that no jobs 'get lost', i.e. a conservation of jobs. The support of an invariant is the set of places with a non-zero weight, e.g. the support of *free + busy = 1* is {*free*,*busy*}.

Given a Petri net which corresponds to a scheduling problem, we find place invariants telling that there is a conservation of tasks and resources. These place invariants are rather trivial. However, we can also focus on place invariants having a support which is a subset of $\{bp_{\langle t,rs \rangle} \mid \langle t, rs \rangle \in dom(PT)\} \cup \{pre_{\langle t,t' \rangle} \mid \langle t, t' \rangle \in PRE\}$. If we find an invariant with such a support, then the weighted-token-sum in these places is constant. Since these places are empty in the initial state, the weighted-token-sum remains zero. Each place in the support of such an invariant will never contain tokens. Therefore, there are no feasible schedules because there are conflicting precedences. Moreover, there is a one-on-one correspondence between conflicting precedences and place invariants with a support which is a subset of $\{bp_{\langle t,rs \rangle} \mid \langle t, rs \rangle \in dom(PT)\} \cup \{pre_{\langle t,t' \rangle} \mid \langle t, t' \rangle \in PRE\}$.

Result 1: *Place invariants can be used to find conflicting precedences.*

To illustrate this result consider the scheduling problem shown in Figure 10. There is one place invariant with a support which is a subset of:
$\{bp_1, bp_2, bp_3, bp_4, pre_{\langle 1,2 \rangle}, pre_{\langle 2,3 \rangle}, pre_{\langle 3,4 \rangle}, pre_{\langle 4,1 \rangle}\}$
It is the place invariant:
$bp_1 + pre_{\langle 1,2 \rangle} + bp_2 + pre_{\langle 2,3 \rangle} + bp_3 + pre_{\langle 3,4 \rangle} + bp_4 + pre_{\langle 4,1 \rangle} = 0$
This invariant shows that the four precedences are in conflict. We have tot remove one of them to obtain a scheduling problem that can be solved.

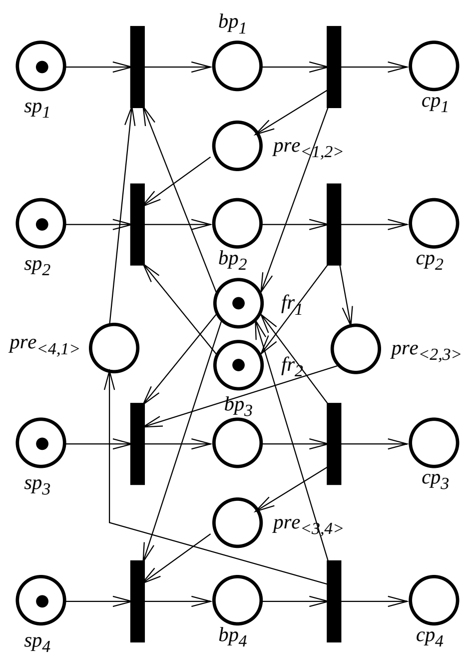

Figure 10: A scheduling problem with conflicting precedences.

We can also use place invariants to find redundant precedence constraints. If we change the direction of the precedence constraint $pre_{\langle 4,1 \rangle}$ (i.e. $pre_{\langle 4,1 \rangle}$ is replaced by $pre_{\langle 1,4 \rangle}$), then we obtain a scheduling problem without conflicting precedences.. However, we find the invariant $bp_1 + pre_{\langle 1,2\rangle} + bp_2 + pre_{\langle 2,3\rangle} + bp_3 + pre_{\langle 3,4\rangle} + bp_4 - pre_{\langle 1,4\rangle} = 0$. (Note the minus sign; this is not a minimal support invariant.) This invariant shows that $pre_{\langle 1,4\rangle}$ is redundant. For details we refer to Peters (1994).

Result 2: *Place invariants can be used to remove redundant precedences.*

Place invariants can be calculated efficiently using linear algebraic techniques (Martinez & Silva, 1982; Murata, 1989). Therefore, standard Petri net tools can be used to find conflicting and/or redundant precedences.

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. There are no transition invariants that hold for a net constructed by following the 'recipe' discussed in Section 4. Therefore, they are not interesting in the context of scheduling.

There are also techniques to verify whether a Petri net is *connected*. A net is said to be connected if and only if each place or transition is connected to any other place or transition, ignoring the direction of the arcs. If a net is not connected it can be decomposed into a number of separate subnets.
If a Petri net which corresponds to a scheduling problem is *not* connected, then we are able to split the scheduling problem into a number of 'independent' scheduling problems.

Result 3: *If possible, we can use the Petri net representation to split a scheduling problem into a number of 'independent' scheduling problems.*

Many other analysis methods have been developed for the analysis of specific structural properties. However, at this point they seem to be irrelevant in the context of scheduling. For more details, we refer to Peters (1994).

5.2 Behavioural properties

There are several methods to analyse the dynamic behaviour of a timed Petri net (Aalst, 1992b; Aalst, 1993; Berthomieu & Diaz, 1991; Carlier, Chretienne & Girault, 1984).

By computing the *reachability graph*, it is possible to analyse all possible firing sequences. Recall that for a net representing a scheduling problem, each of these firing sequences corresponds to a feasible schedule. Therefore, we can use the reachability graph to generate many feasible schedules. Unfortunately, the reachability graph cannot be used to generate all feasible schedules. In fact, we can only generate *eager schedules* (often referred to as list schedules). An eager schedule assigns resources to tasks as soon as possible, i.e. if a task can be executed by a specific resource set and each resource in this resource set is free, then the resource set is allocated to this task and the processing starts immediately.

Result 4: *We can use the reachability graph to find all eager schedules.*

If we consider all schedules generated by the reachability graph with respect to some performance measure, then we are able to determine an optimal eager schedule. However, there may be non-eager schedules surpassing such an optimal eager schedule (see Section 7). If we omit the requirement that transitions fire as soon as possible, then we can use the reachability graph to determine a truly optimal schedule. However, if we omit the eagerness requirement, the reachability graph 'explodes'. In Carlier et al. (1984, 1988) this problem is dealt with for a specific class of scheduling problems.

In the remainder of this section we restrict ourselves to timed Petri nets with eager transitions, i.e. we do not consider non-eager schedules. Nevertheless, for large scheduling problems the reachability graph may still be too large. There are several approaches to (partially) solve this problem. Before discussing some of these approaches, we focus on the construction process of the reachability graph.

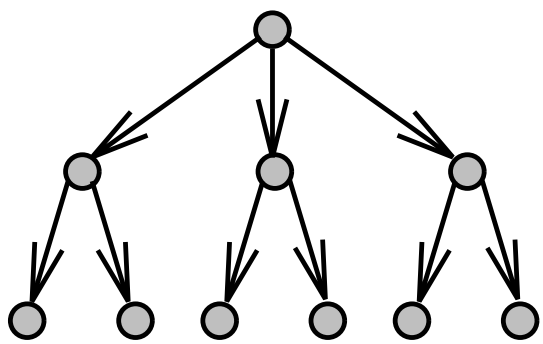

Figure 11: A reachability graph.

The reachability graph of a timed Petri net is constructed as follows. We start with an initial state s. Then we calculate all states reachable from s by firing a transition. For each of these states we calculate the states reachable by firing a transition, etc. Each node in the reachability graph corresponds to a reachable state and each arc corresponds to the firing of a transition (see Figure 11).
One way to reduce the size of the reachability graph is to allow only a limited number of outgoing arcs for each node, i.e. if there are too many successor nodes, we only select a subset of them (randomly).
Another approach is to omit the nodes which are not very 'promising', e.g. if a node corresponds to a partial schedule with a relatively large makespan, we do not consider its successors. We can also omit nodes that correspond to a partial schedule which violates one of the due-dates.
Finally, we can use heuristics to reduce the number of outgoing arcs, e.g. if we can allocate a resource to a large task or a small task, then we select the small task. Note that we can use the priority rules for rule based scheduling (Haupt, 1989). Typical priority rules are: SPT (shortest processing time), MWKR (most work remaining), LWKR (least work remaining), DD (earliest due-date), etc. It is quite easy to extend the timed Petri net model

defined in Section 2 with priorities, i.e. a priority is assigned to each transition. If several transitions are enabled at the same time, then the transition with the highest priority will fire first. If several transitions having equal priorities are enabled at the same time, then any of these transitions may be the next to fire. Extending the timed Petri net model with priorities, facilitates the modelling of priority rules such as SPT, MWKR, LWKR, DD. Moreover, we can still use some of the standard Petri net tools.
If we omit the nodes which are not very 'promising' or use heuristics like the SPT-rule, then we are able to construct a reachability graph of limited size. We can use this reachability graph to find feasible schedules. Note that the makespan of these schedules represents an upperbound for the makespan of the scheduling problem.

Result 5: *By constructing only a part of the reachability graph, we can find upperbounds for the makespan of a scheduling problem.*

It is also possible to find a lowerbound for the makespan of a scheduling problem. Simply remove all fr_r-places and construct the reachability graph. By inspecting the terminal states of the reachability graph, we can deduce a lowerbound for the makespan of the scheduling problem. Although the size of the reachability graph is limited, it may be worthwhile to use the ATCFN analysis method (Aalst, 1992b) to find the same lowerbound.

Result 6: *We can also find a lowerbound for the makespan of a scheduling problem.*

It is also possible to use *simulation* to analyse the dynamic behaviour of a timed Petri net which models a scheduling problem. Such a timed Petri net can be simulated by randomly selecting an enabled transition to be fired. Each subrun results in a terminal state which corresponds to a feasible schedule. In case of deterministic processing times it is not worthwhile to use simulation. However, if we want to test the robustness of a schedule, simulation may be useful.

Result 7: *We can use simulation to test the robustness of a schedule.*

This concludes our presentation of the seven results.

The results presented in this section show that we can use standard Petri-net techniques to analyse a scheduling problem. Nevertheless, these results can also be achieved without using the Petri net formalism. If we model the precedence relation as a directed graph, then result 1 corresponds to the existence of a directed cycle and result 2 corresponds to the existence of a transitive arc. The other results also have their counterparts in the scheduling domain. Results 5 corresponds for example with truncated branch and bound methods and heuristics.
Although most of the results presented in this section can also be achieved with the techniques commonly used in the 'scheduling community', it is important to see that scheduling problems can also be analysed with standard Petri-net tools! Each of the results re-

ported in this section can be applied without any programming. We have developed a tool which automatically translates a scheduling problem into a timed Petri net. We have experimented with two Petri-net based analysis tools: IAT and INA. *IAT* is part of the ExSpect workbench and allows for the calculation of invariants and (condensed) reachability graphs (Aalst, 1992b; Aalst, 1994). (IAT and some other parts of ExSpect have been developed under the supervision of the author of this paper.) *INA* is an analysis tool which allows for many analysis methods. INA can be used to determine more than 40 different properties (Starke, 1992). Note that we use standard Petri net tools without developing new analysis-software!

6 Extensions

In Section 3 we defined what we mean by a scheduling problem. Although Definition 2 is quite general, we made a number of assumptions. However, only few scheduling problems encountered in practise obey each of these assumptions. Therefore, we are interested in the relaxation of some of these assumptions. In this section we show the impact of these relaxations on the corresponding Petri net.

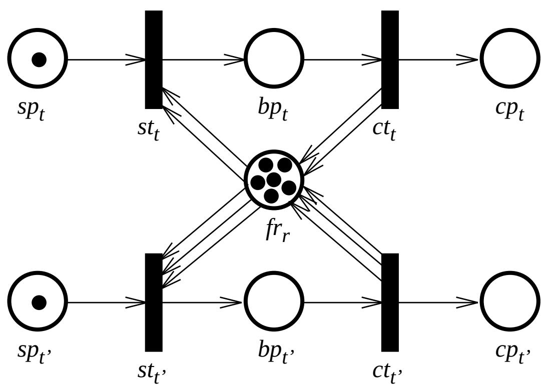

Figure 12: Resource r can process multiple tasks at a time.

First of all, we assumed that each resource can process only one task at a time. If this assumption is dropped, then we have to deal with resources having a specific capacity and tasks requiring only a part of this capacity. It is easy to model this in terms of a Petri net. Consider two tasks t and t' and a resource r. Both tasks have to be processed by r. Resource r has a capacity of 6, task t requires a capacity of 2 and task t' requires a capacity of 3. Figure 12 shows how this can be modelled in terms of a timed Petri net. Initially, place fr_r contains 6 tokens. There are two input arcs from fr_r to st_t indicating that task t requires $2/6$ of the capacity of r, i.e. transition st_t can only fire if there are at least two tokens in place fr_r. Processing t starts with the consumption of two tokens from fr_r (by st_t) and finishes with the production of two tokens for fr_r (by ct_t).

We also assumed that each resource is continuously available for processing. It is easy to introduce 'release times' for resources; initially the token in a place fr_r has a timestamp equal to the release time of the resource r. Dealing with time-windows for the availability of resources is more complicated but not impossible.

If we allow pre-emption, then a task t is no longer represented by the subnet shown in Figure 5. To handle this relaxation we have to split tasks into smaller tasks. Each subtask corresponds to a phase in the processing of task t. A task is allowed to pre-empt the moment it switches from one phase to another.

In Section 3 we assumed that processing times are known and fixed, i.e. the scheduling problem is deterministic. The approach described in this paper can easily be extended to non-deterministic scheduling problems by using another timed Petri net model. There are timed Petri net models with stochastic delays (cf. Marsan, Balbo & Conte (1984, 1986)) or delays described by intervals (Aalst, 1992b; Aalst, 1993; Berthomieu & Diaz, 1991). By mapping the scheduling problem onto such a Petri net model, we can handle problems for which uncertainty is a dominant factor.

The approach presented in this paper allows for many other extensions, e.g. more sophisticated precedence constraints, set-up times, coupling, etc. Moreover, most of the results presented in Section 5 also hold for the relaxations discussed in this section.

7 Case: 10 $\times$ 10

We will use the notorious scheduling problem described by Fisher and Thompson in (1963) to illustrate our approach. This job-shop scheduling problem is concerned with the allocation of 10 machines over 10 jobs each requiring 10 operations, i.e. 10 $\times$ 10 operations have to be processed by 10 machines. Each row in Table 1 corresponds to a

Job id.	M	PT	M	PT	M	PT	M	PT	M	PT	M	PT	M	PT	M	PT	M	PT	M	PT
0	0	29	1	78	2	9	3	36	4	49	5	11	6	62	7	56	8	44	9	21
1	0	43	2	90	4	75	9	11	3	69	1	28	6	46	5	46	7	72	8	30
2	1	91	0	85	3	39	2	74	8	90	5	10	7	12	6	89	9	45	4	33
3	1	81	2	95	0	71	4	99	6	9	8	52	7	85	3	98	9	22	5	43
4	2	14	0	6	1	22	5	61	3	26	4	69	8	21	7	49	9	72	6	53
5	2	84	1	2	5	52	3	95	8	48	9	72	0	47	6	65	4	6	7	25
6	1	46	0	37	3	61	2	13	6	32	5	21	9	32	8	89	7	30	4	55
7	2	31	0	86	1	46	5	74	4	32	6	88	8	19	9	48	7	36	3	79
8	0	76	1	69	3	76	5	51	2	85	9	11	6	40	7	89	4	26	8	74
9	1	85	0	13	2	61	6	7	8	64	9	76	5	47	3	52	4	90	7	45

Table 1: The 10 $\times$ 10 scheduling problem: 10 $\times$ 10 operations have to be processed by 10 machines.

job and lists a sequence of machines (M) and processing times (PT). The first operation required by job 0 has to be processed by machine 0 and the processing time is 29 time units. The second operation is processed by machine 1 and the processing time is 78 time units, etc. The problem is to find a schedule such that the makespan, i.e. the maximal flow-time, is minimal. Although this problem was formulated in 1963, it has defied solution for more than twenty years. In 1989, Carlier & Pinson (1989) proved 930 to be the minimal makespan.

First, we formulate the 10 $\times$ 10 problem in terms of the terminology given in Section 3.

There are 100 tasks, 10 for each job. There are 10 resources, one for each machine. There are 90 precedences, 9 for each job. Each task has a release time equal to 0. Each task requires a specific machine to be processed and the processing times are as indicated in Fisher & Thompson (1963).
Then, we map the scheduling problem onto a timed Petri net (see Definition 3). We have used the Petri net based tool *ExSpect* (Aalst, 1994) to construct this net automatically. The corresponding timed Petri net contains 400 places and 200 transitions.

We will use *IAT*, one of the analysis tools of ExSpect (Aalst, 1992b; Aalst, 1994), to analyse the constructed timed Petri net. IAT is based on a number of Petri net based analysis techniques (e.g. place and transition invariants, reachability graphs, reduction techniques, etc.).
The constructed net is connected, i.e. the 10×10 problem cannot be split into a number of smaller problems (see Section 5). Moreover, there are no place invariants with a support which is a subset of $\{bp_{\langle t,rs\rangle} \mid \langle t,rs\rangle \in dom(PT)\} \cup \{pre_{\langle t,t'\rangle} \mid \langle t,t'\rangle \in PRE\}$, i.e. there are no conflicting precedences. These results are not very surprising for this well-structured scheduling problem. Moreover, in this case we are much more interested in schedules with a small makespan.
It is very easy to calculate an upper bound for the minimal makespan of the 10×10 problem; simply generate a reachability graph where each node is allowed to have only one successor. In this case we find one terminal state. This state corresponds to a feasible schedule. The first upper bound we found was 1190, IAT calculates this upper bound in 15 seconds. If we had been able to calculate the entire reachability graph we could have calculated an optimal non-eager schedule. Unfortunately, in this case the reachability graph is too large to construct. We also used priority rules to obtain a smaller reachability graph. This resulted in smaller upper bounds. However, even the best priority rules we have tested result in schedules with a makespan of more than 1100.
We used the ATCFN analysis method (Aalst, 1992b) to calculate a lower bound of 691 for the makespan of any feasible schedule. This takes about 14 seconds.

We also tested an approach which adds extra precedence constraints. This approach resulted in a schedule with a makespan equal to 1023. For any two tasks t and t' we added the precedence constraint that t has to complete before t' starts if and only if (1) there is more work remaining for the job where t belongs to than the work remaining for the job where t' belongs to and (2) the processing time of t is rather small. Without going into details, we postulate that this approach outachieves the priority rules used in rule based scheduling. However, it does not lead to schedules having a makespan close to 930. It takes about 22 seconds to calculate the schedule with a makespan of 1023.
Note that we obtained these results by using standard Petri net tools, i.e. without developing special purpose algorithms or software.

8 Conclusion

The approach presented in this paper shows that it is possible to model many scheduling problems in terms of a timed Petri net. In fact, we have formulated a recipe for mapping scheduling problems onto timed Petri nets. This recipe shows that the Petri net formalism can be used to model tasks, resources and precedence constraints.

By mapping a scheduling problem onto a timed Petri net, we are able to use Petri net theory to analyse the scheduling problem. We can use Petri net based analysis techniques to detect conflicting precedences, determine lower and upper bounds for the minimal makespan, etc. By inspecting (parts of) the reachability graph, we can generate many feasible schedules. Although it is likely that these analysis techniques will never beat the scheduling algorithms described in literature, we can use standard Petri net tools without developing new software.

Last but not least, we hope that the link between scheduling and Petri nets will stimulate further research in scheduling and Petri net analysis. On the one hand, Petri net based analysis techniques have to be improved to deal with the computational complexity of scheduling problems. On the other hand, modelling scheduling problems in terms of timed Petri nets will bring new scheduling problems not considered by existing solution approaches.

References

AALST, W.M.P. VAN DER (1992a), Modelling and Analysis of Complex Logistic Systems, in: H.J. Pels and J.C. Wortmann (eds.), *Integration in Production Management Systems*, IFIP Transactions B-7, Elsevier Science Publishers, Amsterdam, 277–292.

AALST, W.M.P. VAN DER (1992b), *Timed coloured Petri nets and their application to logistics*, Ph.D. thesis, Eindhoven University of Technology, Eindhoven.

AALST, W.M.P. VAN DER (1993), Interval Timed Coloured Petri Nets and their Analysis, in: M. Ajmone Marsan (ed.), *Application and Theory of Petri Nets 1993*, Lecture Notes in Computer Science 691, Springer-Verlag, Berlin, 453–472.

AALST, W.M.P. VAN DER (1994), Putting Petri nets to work in industry, *Computers in Industry* **25**, 45–54.

BAKER, K.R. (1974), *Introduction to Sequencing and Scheduling*, Wiley & Sons.

BERTHOMIEU, B. AND M. DIAZ (1991), Modelling and verification of time dependent systems using Time Petri Nets, *IEEE Transactions on Software Engineering* **17**, 259–273.

CARLIER, J. AND P. CHRETIENNE (1988), Timed Petri net schedules, in: G. Rozenberg (ed.), *Advances in Petri Nets 1988*, Lecture Notes in Computer Science 340, Springer-Verlag, Berlin, 62–84.

CARLIER, J., P. CHRETIENNE, AND C. GIRAULT (1984), Modelling scheduling problems with Timed Petri Nets, in: G. Rozenberg (ed.), *Advances in Petri Nets 1984*, Lecture Notes in Computer Science 188, Springer-Verlag, Berlin, 62–82.

CARLIER, J. AND E. PINSON (1989), An algorithm for solving the job-shop problem, *Management Science* **35**, 164–176.

CHRETIENNE, P. (1983), *Les réseaux de petri temporisés*, Ph.D. thesis, Univ. Paris VI, Paris.

FISHER, H. AND G.L. THOMPSON (1963), Probabilistic learning combinations of local job-shop scheduling rules, in: J.F. Muth and G.L. Thompson (eds.), *Industrial Scheduling*, Prentice-Hall, Englewood Cliffs, 225–251.

FRENCH, S. (1982), *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop*, Wiley & Sons.

GAO, G.R., Y.B. WONG, AND Q. NING (1991), A timed Petri-net model for loop scheduling, *Proceedings of the 12th International Conference on Applications and Theory of Petri Nets*, Gjern, 22–41.

HAUPT, R. (1989), A survey of priority rule-based scheduling, *OR Spectrum* **11**, 3–16.

HILLION, H.P. AND J.P PROTH (1989), Performance Evaluation of Job-Shop Systems Using Timed Event Graphs, *IEEE Transactions on Automatic Control* **34**, 3–9.

JENSEN, K. (1992), *Coloured Petri Nets. Basic concepts, analysis methods and practical use.*, EATCS monographs on Theoretical Computer Science, Springer-Verlag, Berlin.

JENSEN, K. AND G. ROZENBERG (eds.) (1991), *High-level Petri Nets: Theory and Application*, Springer-Verlag, Berlin.

LAWLER, E.L., J.K. LENSTRA, A.H.G. RINNOOY KAN, AND D.B. SHMOYS (1993), Sequencing and scheduling: Algorithms and complexity, in: S.C. Graves, A.H.G. Rinnooy Kan, and P. Zipkin (eds.), *Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory*, North-Holland, Amsterdam.

MARSAN, M. AJMONE, G. BALBO, AND G. CONTE (1984), A Class of Generalised Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems, *ACM Transactions on Computer Systems* **2**, 93–122.

MARSAN, M. AJMONE, G. BALBO, AND G. CONTE (1986), *Performance Models of Multiprocessor Systems*, The MIT Press, Cambridge.

MARTINEZ, J. AND M. SILVA (1982), A simple and fast algorithm to obtain all invariants of a generalised Petri Net, in: C. Girault and W. Reisig (eds.), *Application and theory of Petri nets : selected papers from the first and the second European workshop*, Informatik Fachberichte 52, Berlin, Springer-Verlag, Berlin, 301–310.

MURATA, T. (1989), Petri Nets: Properties, Analysis and Applications, *Proceedings of the IEEE* **77**, 541–580.

PETERS, N. (1994), Analysis of scheduling problems by means of INA/ExSpect, Master's thesis, Eindhoven University of Technology, Eindhoven.

PETRI, C.A. (1962), *Kommunikation mit Automaten*, Ph.D. thesis, Institut für instrumentelle Mathematik, Bonn.

PINEDO, M. (1995), *Scheduling : theory, algorithms, and systems*, Prentice-Hall, Englewood Cliffs.

RAMAMOORTHY, C.V. AND G.S. HO (1980), Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets, *IEEE Transactions on Software engineering* **6**, 440–449.

REISIG, W. (1985), *Petri nets: an introduction*, Prentice-Hall, Englewood Cliffs.

STARKE, P.H. (1992), *INA: Integrierter Netz Analysator, Handbuch*.

WATANABE, T. AND M YAMAUCHI (1993), New priority-lists for scheduling in timed Petri nets, in: M. Ajmone Marsan (ed.), *Application and Theory of Petri Nets 1993*, Lecture Notes in Computer Science 691, Springer-Verlag, Berlin, 493–512.