INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité : AUTOMATIQUE, PRODUCTIQUE

préparée au laboratoire VERIMAG

dans le cadre de **L'Ecole Doctorale ELECTRONIQUE, ELECTROTECHNIQUE, AUTOMATIQUE, TELECOMMUNICATION, SIGNAL**

présentée et soutenue publiquement

par

Yasmina Abdeddaïm
le 28 Novembre 2002

Titre

Modélisation et Résolution de Problèmes d'Ordonnancement à l'aide d'Automates Temporisés

Directeur de thèse

Oded Maler

JURY

M. Jean Della Dora, President
M. Ed Brinksma, Rapporteur
M. Kim Larsen, Rapporteur
M. Oded Maler, Directeur de thèse
M. Claude Le Pape, Examinateur
M. Stavros Tripakis, Examinateur

Remerciements

Je remercie Mr Oded Maler, mon directeur de thèse, il a été un encadreur exceptionnel, tant par l'originalité de ses idées que par sont engagement dans mon travail, il a été pour moi l'encadreur idéal.

Je remercie Mr Jean Della Dora, Mr Ed Brinksma, Mr Kim Larsen, Mr Claude Le pape et Mr Stavros Tripakis d'avoir participé à mon jury de thèse.

Je remercie Mr Joseph Sifakis directeur du laboratoire VERIMAG pour avoir su diriger un laboratoire où règnent toutes les bonnes conditions de travail et où on retrouve un grand mélange culturel très enrichissant.

Je remercie Mr Eugene Asarin pour tout le temps qu'il m'a consacré durant nos longues discussions, pour sa grande pédagogie et sa modestie.

Je remercie Marius Bozga connu à VERIMAG pour être toujours disponible à donner un coup de main et des explications très claires.

Je remercie Stavros Tripakis pour ses discutions scientifiques et générales toujours pleines de convictions.

Un grand merci aux chercheurs et enseignants de VERIMAG et plus spécialement Sergio Yovine, Paul Caspi et Saddek Bensalem.

Merci à Thao Dang ma soeur de VERIMAG, Gordon Pace pour toutes les journées passées ensemble, Christos Kloukinas et ses silences expressifs, Moez Mahfoudh et ses histoires abracadabrantes, Radu Iosif qui s'est très vite intégré, Catalin Dima qui a supporté mes taquineries.

Merci à mes amis et à mes rencontres de Grenoble, je ne peux tous les citer, qui m'ont fait réfléchir sur d'autres sujets que mon sujet de thèse.

Merci à mes amis d'Alger qui ont su me rester fidèles.

Je tiens à remercier ma famille, mes parents qui m'ont donné un grand coup de pouce, mon frère Said qui a été pour moi d'un grand secours durant toutes les étapes de ma thèse et qui a su me conseiller avec son grand calme communicatif, ma soeur Amina et son grand sourire, je te promets que je vais finir par trouver du temps pour venir te voir.

Merci Zahir.

Contents

Chapter 1

Introduction

1.1 Contribution of the Thesis

This thesis develops a new methodology for posing and solving scheduling problems. The essence of this approach is to model the system as a timed automaton where schedules correspond to paths in the automaton and optimal schedules correspond to shortest paths.

The methods that we propose are inspired by several existing bodies of knowledge. The first is algorithmic verification methodology whose goal is to prove some properties concerning all runs of a discrete transition system (automaton). Verification algorithms are based, this way or another, on graph algorithms that explore paths in the transition graph. On slightly richer models, where numerical weights are assigned to arcs or to nodes of the graph, one can formulate and solve shortest-path algorithms, that is, find the minimal cost path between two given nodes. Our goal is to use such algorithms to find optimal schedules, but in order to do so we need to find a way to extend shortest path algorithms to problems where the cost associated with a path is the time elapsed from beginning to end.

In the first part of the document, we start by a short survey of the job-shop problem (Chapter 2) and the techniques traditionally used for its solution. In the Chapter 3 we give a survey of the basic algorithms for exploring directed graphs. In Chapter 4 we review a naive approach for expressing passage of time using weighted automata, then we move on to timed automata.

The second part concerns our approach. We first model the classical job-shop problem (Chapter 5), and then extend the model to treat preemption (Chapter 6), partially-ordered tasks (Chapter 7) and scheduling problems with temporal uncertainty (Chapter 8). For all these problems we develop algorithms, implement them and test their performance on benchmark examples.

The thesis is a contribution both to the theory and practice of scheduling and to the analysis of timed automata.

1.2 Related Work

This work can be viewed in the context of extending verification methodology in two orthogonal directions: from *verification* to *synthesis* and from *qualitative* to *quantitative* evaluation of behaviors. In verification we check the existence of certain paths in a *given* automaton, while in synthesis we have an automaton in which not all design choices have been made and we can

remove transitions (and hence make the necessary choices) so that a property is satisfied. If we add a quantitative dimension (in this case, the duration of the path), verification is transformed to the evaluation of the worst performance measure over all paths, and synthesis into the restriction of the automaton to one or more optimal paths.

The idea of applying synthesis to timed automata was first explored in [WH92]. An algorithm for safety controller synthesis for timed automata, based on operation on zones was first reported in [MPS95] and later in [AMP95], where an example of a simple scheduler was given, and in [AMPS98]. This algorithm is a generalization of the verification algorithm for timed automata [HNSY94, ACD93] used in Kronos [Y97, BDM$^+$98]. In these and other works on treating scheduling problems as synthesis problems for timed automata, such as [AGP99], the emphasis was on yes/no properties, such as the existence of a feasible schedule, in the presence of an uncontrolled adversary.

A transition toward quantitative evaluation criteria was made already in [CY91] where timed automata were used to compute bounds on delays in real-time systems and in [CCM$^+$94] where variants of shortest-path problems were solved on a timed model much weaker than timed automata. To our knowledge, the first quantitative synthesis work on timed automata was [AM99] in which the following problem has been solved: "given a timed automaton with both controlled and uncontrolled transitions, restrict the automaton in a way that from each configuration the worst-case time to reach a target state is minimal". If there is no adversary, this problem corresponds to finding the shortest path. Due to the presence of an adversary, the solution in [AM99] employs backward-computation (dynamic programming), i.e. an iterative computation of a function $h : Q \times \mathcal{H} \rightarrow \mathbb{R}_+$ such that $h(q, \mathbf{v})$ indicates the minimal time for reaching the target state from$(q, \mathbf{v})$. The implementation of the forward algorithm developed in this thesis can be viewed as iterating with a function h such that $h(q, \mathbf{v})$ indicates the minimal time to reach $(q, \mathbf{v})$ from the initial state. The reachable states in the augmented clock-space are nothing but a relational representation of h.

Around the same time, in the framework of the VHS (Verification of Hybrid systems) project, a simplified model of a steel plant was presented as a case-study [BS99]. The model had more features than the job-shop scheduling problem such as upper-bounds on the time between steps, transportation problems, etc. A. Fehnker proposed a timed automaton model of this plant from which feasible schedules could be extracted [F99]. Another work in this direction was concerned with another VHS case-study, a cyclic experimental batch plant at Dortmund for which an optimal dynamic scheduler was derived in [NY00].

The idea of using heuristic search is useful not only for shortest-path problems but for verification of timed automata (and verification in general) where some evaluation function can guide the search toward the target goal. These possibilities were investigated recently in [BFH$^+$01a] on several classes of examples, including job-shop scheduling problems, where various search procedures and heuristics were explored and compared.

In [NTY00] it was shown that in order to find shortest paths in a timed automaton, it is sufficient to look at acyclic sequences of symbolic states (a fact that we do not need due to the acyclicity of job-shop automata) and an algorithm based on forward reachability was introduced. A recent generalization of the shortest path problem was investigated by [BFH$^+$01b] and [ATP01], in a model where there is a *different* price for staying in any state and the total cost associated with the run progresses in different slopes along the path. It has been proved that the problem of finding the path with the minimal cost is solvable.

Part I

Background

Chapter 2

Job Shop Scheduling

The *job shop problem* is one of the most popular problems in scheduling theory. On one hand it is very simple and intuitive while on the other hand it is a good representive of the general domain as it exhibits the difficulty of combinatorial optimization. The difficulty is both theoretical (even very constrained versions of the problem are NP-hard) and practical (an instance of the problem with 10 jobs and 10 machines, proposed by Fisher and Thompson [F73a], remained unsolved for almost 25 years, in spite of the research effort spent on it). In the rest of the section we give an informal presentation of the problem and mention some of the methods suggested in the past for its solution.

2.1 The problem

A job shop problem consists of a finite set $\mathcal{J} = \{J^1, \ldots, J^n\}$ of jobs to be processed on a finite set $\mathcal{M} = \{m_1, \ldots, m_m\}$ of machines. Each job J^i is a finite sequence of steps to be executed one after the other, where each step is a pair of the form (m, d), $m \in M$ and $d \in \mathbb{N}$, indicating the required utilization of machine m for time duration d. Each machine can process at most one step at a time, and, due to precedence constraints, at most one step of each job may be processed at any time. Steps cannot be preempted once started.

The objective is to determine the starting times for each step in order to minimize the total execution time of all jobs, i.e the time the last step terminates. This problem is denoted in the scheduling community as $J||C_{max}$ where C_{max} is the maximum completion time, called *makespan*.

Example 1.1 Consider $\mathcal{M} = \{m_1, m_2, m_3\}$ and three jobs J^1, J^2, J^3 to be scheduled on these machines. The job J^1 consists of 3 steps, the first lasts 3 time units and must be carried out on the machine m_1, the second lasts 6 time units on machine m_3 and so on.

$$J^1 : (m_1, 3), (m_3, 6), (m_2, 5)$$
$$J^2 : (m_1, 2), (m_1, 4), (m_3, 7)$$
$$J^3 : (m_3, 1), (m_2, 5), (m_1, 6)$$

Two schedules S_1 and S_2 are depicted in Figure 2.1 The length of S_1 is 17 while the length of S_2 is 16, and it is the optimal schedule. The two schedules are represented using Gantt diagram, showing the evolution of each job on the machines.

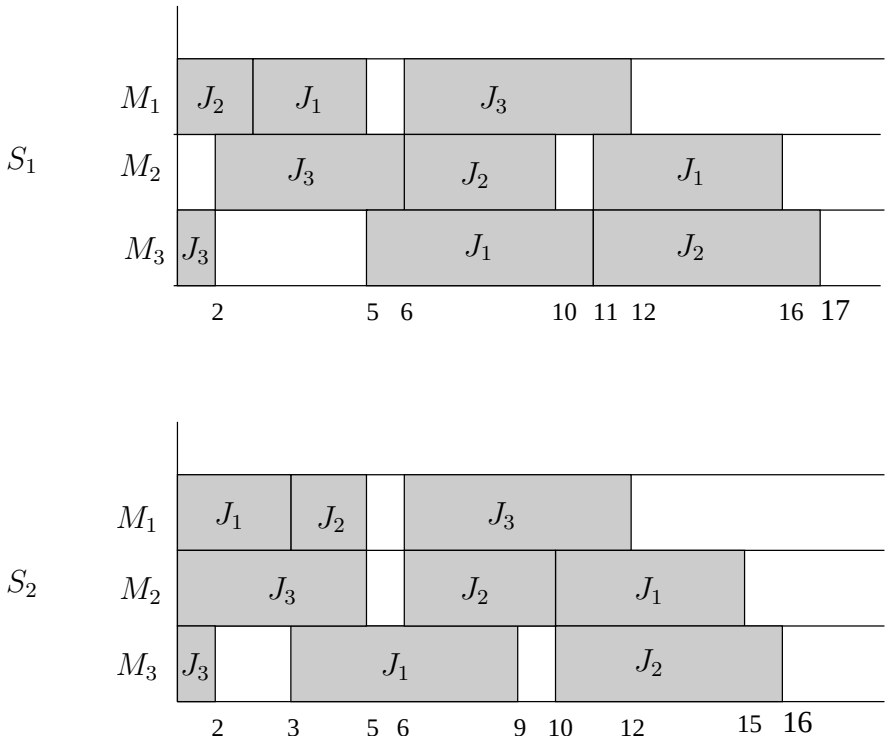


Figure 2.1: Two schedules S_1 and S_2 represented in a Gantt diagram

2.2 Disjunctive Graph Representation

In the lack of resource conflicts (when every job uses a distinct set of machines) the optimal schedule is achieved by letting every job execute its steps as soon as possible, that is, to start immediately with the first step and move to the next step as soon the previous step terminates. In this case the minimal makespan is the maximal (over all jobs) sum of step durations. The difficult (and interesting) part of the problem comes with conflicts: a job might want to start a step but the corresponding machine is occupied by another job. Here the optimal schedule need not be based on the greedy principle "start every step as soon as it is enabled" because it might be globally better to wait and leave the machine free for a later step of another job. It is this type of decisions that distinguishes one schedule from another as can be seen in an intuitive manner using the *disjunctive graph* model, introduced by Roy and Sussman [RS64].

A disjunctive graph associated with a job-shop problem is a graph $G = (V, W, A, E)$ where V is a set of nodes corresponding to the steps of the problem with two additional (fictitious) nodes "start" and "end". The positive weight $W(v)$ associated with each node v is equivalent to the duration the corresponding step. The precedence constraints between the steps of the same job are represented by a set A of directed edges such that $(v, v') \in A$ indicates that step v is an immediate predecessor of step v'. The resource conflicts are represented by a set E of undirected edges, such that whenever step v and step v' use the same machine, both $(v, v') \in E$ and $(v', v) \in E$. Clearly, E is an equivalence relation and let us denote by E_m the subset of E corresponding to machine m. An *orientation* of E is a subset $E' \subseteq E$ such that for every machine E'_m is a linear order. Every orientation can be seen as defining a *priority relation* between steps requiring the same machine. Two examples of orientation of the graph appears in Figure 2.2 where job J^2 has priority over job J^1 on machine m_1 in the orientation corresponding to the schedule S_1 and the opposite on the orientation of the schedule S_2.

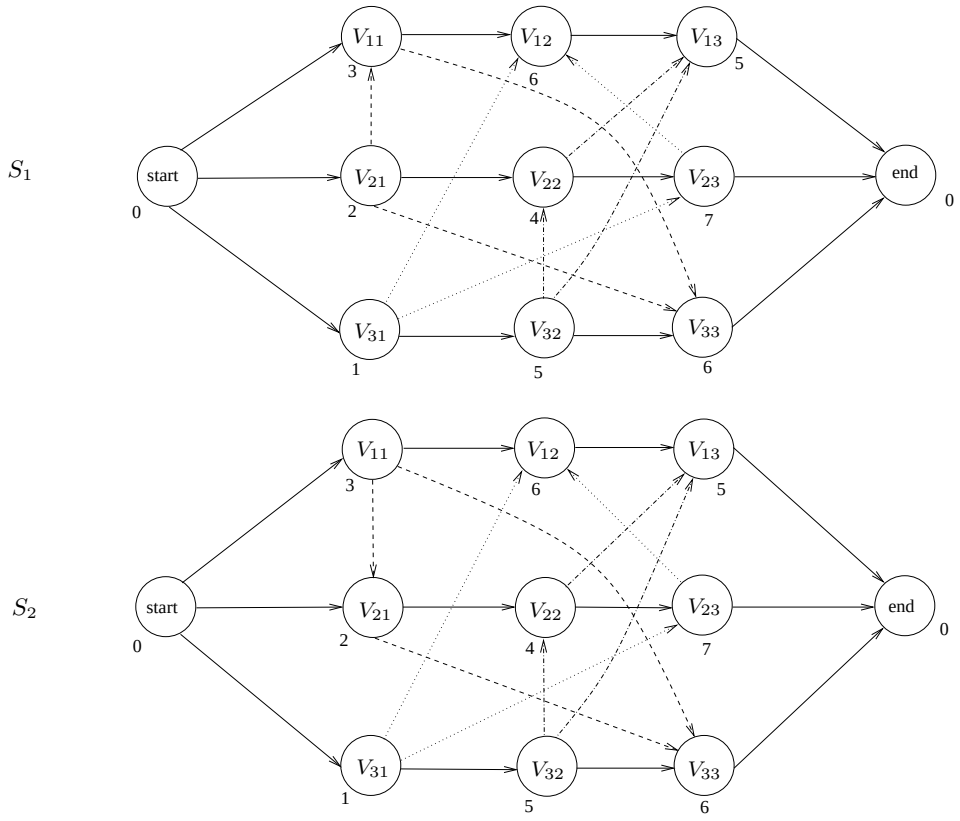


Figure 2.2: The two orientations corresponding to the schedules S_1 and S_2 where V_{ij} is the node corresponding to the step j of the job i.

For every given orientation, the length of the longest path from "start" to "end" is equal to the length of the shortest schedule among all the schedules that satisfy the corresponding priority relation. So finding the optimal schedule reduces to finding the priority relation whose associated orientation leads to the minimal longest path. There are finitely many such priority relations, hence the problem is solvable, however their number is exponential which makes the problem intractable. Note that some partial choices of priorities can make other priorities redundant, for example, if one job uses m at an early step and another job uses machine m toward the end, it might be the case that the priority relation between them need not be stated explicitly.

2.3 Constrained Optimization Formulations

An alternative formulation of the problem is as a constrained linear optimization problem where the combinatorial difficulty is manifested by the *non-convexity* of the set of feasible solutions. For the purpose of this formulation let us write a job J^i as a sequence of steps

$$(m_{i1}, d_{i1}), (m_{i2}, d_{i2}), \ldots, (m_{im}, d_{im}),$$

let t_{ij} denote the start time of step j of job i and let T be the global termination time.

The job shop problem can be formulated mathematically in the following form:

$$\left\{\begin{array}{l}\text{Minimize } T \text{ where } \forall J^i \;\; t_{im} \leq T \;\; (T \text{ is the maximal makespan})\\ \text{subject to :}\\ \quad \forall J^i \in \mathcal{J} \; \forall k \in \{1 \ldots m\} \qquad t_{ik} \geq 0\\ \quad \forall J^i \in \mathcal{J} \qquad\qquad t_{ik} - t_{ih} \geq d_{ih} \quad \text{if } (m_{ih}, d_{ih}) \text{ precedes } (m_{ik}, d_{ik})\\ \quad \forall J^i \in \mathcal{J} \; \forall J^j \in \mathcal{J} \text{ if } (m_{ik} = m_{jh})\\ \qquad\qquad t_{jk} - t_{ik} \geq d_{ik} \qquad (m_{ik}, d_{ik}) \text{ precedes } (m_{jh}, d_{jh})\\ \qquad\qquad \vee\\ \qquad\qquad t_{ik} - t_{jk} \geq d_{jk} \qquad (m_{jh}, d_{jh}) \text{ precedes } (m_{ik}, d_{ik})\end{array}\right.$$

As one can see, the question of priority appears now in the form of disjunction. If we transform the constraints into a disjunctive normal form, every disjunct will indeed correspond to a priority relation under which the problem transforms into a trivial linear programming problem. An attempt to avoid the enumeration of priority relation and stay within the framework of linear programming is to transform the problem into mixed integer linear programming (MILP) format of Manne [Ma96].

A MILP problem is a linear program where some of the variables are integers. In the job shop problem the integer variables are binary and are used to model the disjunctive constraints.

$$\left\{\begin{array}{l}\text{Minimize } T \text{ where } \forall J^i \;\; t_{im} \leq T \;\; (T \text{ is the maximal makespan})\\ \text{subject to :}\\ \quad \forall J^i \in \mathcal{J} \; \forall k \in \{1 \ldots m\} \qquad t_{ik} \geq 0\\ \quad \forall J^i \in \mathcal{J} \;\; \forall J^j \in \mathcal{J} \;\; \forall m_k \in \mathcal{M} \qquad y_{ijk} \in \{0, 1\}\\ \quad \forall J^i \in \mathcal{J} \qquad\qquad t_{ik} - t_{ih} \geq d_{ih} \quad \text{if } (m_{ih}, d_{ih}) \text{ precedes } (m_{ik}, d_{ik})\\ \quad \forall J^i \in \mathcal{J} \;\; \forall J^j \in \mathcal{J} \text{ if } (m_{ik} = m_{jh})\\ \qquad\qquad t_{jk} - t_{ik} + K(1 - y_{ijk}) \geq d_{ik}\\ \qquad\qquad t_{ik} - t_{jk} + K(y_{ijk}) \geq d_{jk}\\ \qquad\qquad \text{K is a big number}\end{array}\right.$$

Even for the more compact mathematical formulations a large number of constraint are required [Ma96] and the number of integer variables grows exponentially [B59]. Giffer and Thompson [HLP93] mention that integer programs have not led to practical methods of solutions while French [GT60] expresses the view that an integer programming formulation of scheduling problems is computationally infeasible.

The best results that has been obtained using mathematical formulation are due to Lagrangian relaxation (LR) approaches [F73b, Fr82, V91, FT63, KSSW95] and decomposition methods [A67, **?**, DR94, M60] The results indicate that solutions are usually of poor quality.

2.4 Enumerative Methods

Since the embedding of the problem in the continuous optimization framework does not seem to work, most approaches are based on enumeration, a search in the space of feasible solutions. In order to avoid an exhaustive search, enumerative methods use clever elimination techniques that reduce the number of solutions that need to be explored. The general framework for such a search is called *Branch and Bound* (BB) and is best explained using the following example.

Suppose our search space consists of all 6 priority relations (permutations) over $\{1, 2, 3\}$. We can create the permutations incrementally, for example decide first whether $1 \prec 2$ or $2 \prec 1$. The first choice amounts to choosing the subset $\{123, 132, 312\}$ of the possible permutations, and each permutation corresponds to a leaf in a search tree such as the one in Figure 2.3. A crucial component of BB is the ability to evaluate the quality of a partial choice (non-leaf node)

by an over-approximation. The general idea is then to choose one or more full branches in the search tree and evaluate them. This gives an upper-bound (if the problem is minimization) on the quality of the solution. Then a search can be conducted on the tree, cutting branches whose estimated evaluation is worse than the upper-bound.

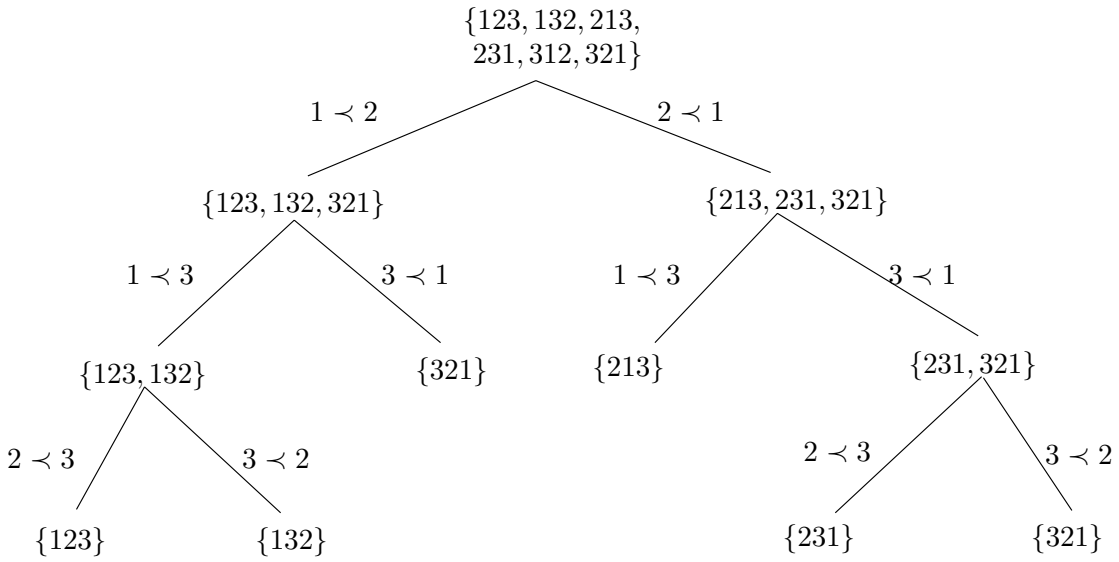


Figure 2.3: A search in the space of priority relations

The first applications of BB to the job shop problem was proposed by Balas [B69], followed by many others, including Carlier [C82], Carlier and Pinson [CPP92], Appelgate and Cook [**?**], Brucker and al. [BJS94], Perregaard and Clausen [PC95], Boyd and Burlingame [BB96] and Martin [Ma96]. Although the computational study indicates that improvements have been achieved by BB methods, this is mainly attributed to improvement in computer technology rather than to the techniques used.

2.5 Approximation methods

Although approximation methods do not guarantee optimal solutions, they can attain near optimal solutions within moderate computing time and are adequate to large problems. Main categories of approximation techniques are: priority dispatch rules, bollelneck based heuristics, Constraint Satisfaction and local search methods.

Approximation applied to job shop problem were first developed on the basis of *Priority Dispatching Rule* (PDR). In this method, at each step all the operations which are available to be scheduled are assigned a priority and the operation with the highest priority is chosen to be sequenced. Usually several runs of PDRs are made in order to achieve valid results. The best results due to this technique are those of Sabuncuoglu and Bayiz [SB97]. However the deviations from the optimum are still high, the results suggest that PDRs are more suitable as an initial solution technique.

The *Shifting Bottleneck* (SB) procedure decomposes the job shop problem into a series of single-machine subproblems. It schedules the machines one by one and focuses on bottleneck machines. One example of such approach is the Shifting Bottleneck Procedure of Adams and all. [ABZ88] A general weakness of the SB approaches is the level of programmer sophistication required.

Constraint Satisfaction techniques aim at reducing the effective size of the search space by applying constraints that restrict the order in which variables are selected and the sequence in which possible values are assigned to each variable. After a value is assigned to a variable any inconsistency arising is removed. These techniques has been applied to solve the job-shop problem in [BPN95, CY94, NA96, PT96, NP98].

Chapter 3

Discrete Verification and Graph Algorithms

3.1 Reachability

The domain of verification is concerned with proving that systems, such as computer programs or digital circuits, behave as required and contain no bugs under all conceivable circumstances. Typically each component of such a system is modeled as an automaton, a system with a finite set of states and transitions between states that corresponds to events or actions performed by the component. In this framework a job in a job-shop problem can be modeled by an automaton whose states represent the progress of the job along its sequence of steps, and transitions correspond to the initiation and termination of steps. The automaton characterizing a whole system is obtained by composing the automata its components[1] and obtaining an automaton whose states are elements of the Cartesian product of the state sets of the components. The set of all possible behaviors of the system is thus represented by the set of all paths in this transition graph, whose size grows exponentially with the number of components.

So technically, the problem of verification can be reduced to the problem of checking the existence of certain paths in a *directed graph*, the transition graph of the automaton. The simplest type of properties to which verification is applied are *reachability* properties, properties that are verified by a path depending on whether or not it visits some specific states. In some cases we want to check that all behaviors avoid a set of "bad" states (for example, a state where the same machine is given to two jobs), and in others we do not want behaviors to get stuck but rather to proceed to some final state (for example, a state where all jobs have terminated). Such properties are examples of *safety* and *liveness* properties, respectively. The algorithmic approach to verification, also known as *model-checking*, uses graph algorithms in order to explore the paths in the automaton and to show that they satisfy these and more complex properties.

In the sequel we will survey some of the basic algorithms for exploring directed graphs, starting with those that can solve reachability problems. Later, by assigning numerical weights to the automaton transitions, we associate real numbers (length, cost) with runs and search for runs that are optimal in this sense, using shortest path algorithms. Automata and directed graphs describe essentially the same objects and we will use the terminology and notation of automata theory.

Definition 1 (Finite State Automata) *A finite state automaton is a pair $\mathcal{A} = (Q, \delta)$ where Q is a finite set of states, and $\delta \subseteq Q \times Q$ is a transition relation. A finite run of the automaton*

[1] There are many forms of compositions depending of the mode of interaction between the components and other features that are beyond the scope of this thesis.

starting from a state q_0 *is a sequence of states*

$$q_0 \to q_1 \to \ldots \to q_k$$

such that $(q_i, q_{i+1}) \in \delta$ *for every* i, $0 \leq i \leq k-1$.

In graph-theoretic terminology states are *vertices*, or *nodes* transitions are *edges* or *arcs* and runs are *paths*. We use $Succ(q)$ and $Pre(q)$ to denote the sets of successors and predecessor of a state q:

$$Succ(q) = \{q' : (q, q') \in \delta\}$$

and

$$Pre(q) = \{q' : (q', q) \in \delta\}.$$

A useful concept for visualizing all the possible runs of the automaton starting from a state q is the *q-unfolding* of the automaton, a tree constructed by starting at q, adding nodes and transitions for each of its successors, and repeating the procedure recursively for these nodes. If the automaton contains cycles (paths leading from a state to itself) its unfolding will be infinite. Otherwise it is finite but its size can be exponential in the size of the automaton. An automaton is depicted in Figure 3.1 and an initial part of its unfolding is shown in Figure 3.2.

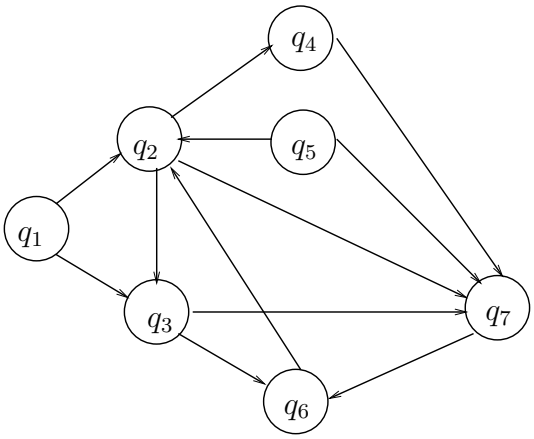


Figure 3.1: An automaton $\mathcal{A} = (Q, \delta)$

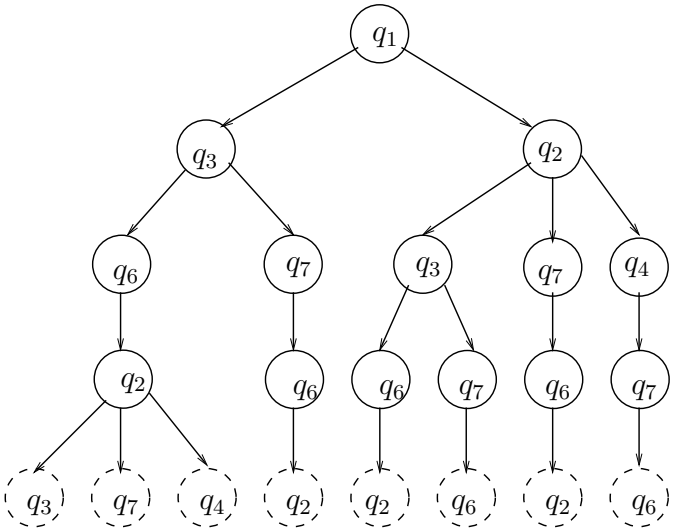


Figure 3.2: The q_1-unfolding of the automaton of Figure 3.1

We will now demonstrate the various approaches to graph searching in order to solve the following reachability problem: "given an automaton and two states q and q', is there a path leading from q to q'". Clearly the answer to this question is positive if the unfolding of the automaton starting at q contains a node labeled by q'. The simplest algorithm for doing it is the breadth-first search (BFS) algorithm described below. The algorithm maintains two data-structures, a set of explored states whose successors have already been encountered and an ordered list of waiting states that need to be explored (these are sometimes called the *frontier* of the search). The algorithm terminates either by reaching q' and answering "yes" or by computing all the states reachable from q without reaching q'. Termination of the algorithm is guaranteed by the finiteness of Q.

Algorithm 1 (Reachability-BFS $(\mathcal{A}, q, q')$**)**

Waiting := $\{q\}$;
Explored:=$\{\}$;
Reached := no;
while *(Reached = no and Waiting* $\neq \emptyset$*)* **do**
begin
  Pick first $v \in$ *Waiting;*
  if *(*$v \notin$ *Explored)* **then**
  begin
    for *every* u *such that* $u \in Succ(v) \wedge u \notin$ *Explored* **do**
      if *(*$u = q'$*)* **then**
        Reached := yes;
      else
        Insert u *into the end of Waiting;*
    Insert v *into Explored;*
  end
  Remove v *from Waiting;*
end
return(Reached);

Note that BFS explores the unfolding of the automaton according to levels, that is, the order in which it explores the nodes of the tree is consistent with their distance from the root, where "distance" here means the number of transitions. If, in addition to the yes/no answer, we want to obtain a path when it exists, we can modify the algorithm by storing with each inserted node the path that has led to it, i.e. inserting $v \cdot u$ into Waiting instead of inserting only u.

While the BFS algorithm advances "in parallel" over all paths, the depth-first search (DFS) algorithm attempts to reach q' along a path and then move to another one. Technically the difference between the algorithms is in the place where new nodes are added to the waiting list, at the end (BFS) or at the beginning (DFS). In other words, the list is FIFO (first in first out) in BFS and LIFO (last in first out) in DFS.

Algorithm 2 (Reachability-DFS $(\mathcal{A}, q, q')$**)**

Waiting:= $\{q\}$*;*
Explored:=$\{\}$*;*
Reached := no;
while *(* $Reached = no$ *and* $Waiting \neq \emptyset$*)* **do**
begin
  Pick first $v \in Waiting$*;*
  if *(*$v \notin Explored$*)* **then**
  begin
    for *every* u *such that* $u \in Succ(v) \wedge u \notin Explored$ **do**
      if *(*$u = q'$*)* **then**
        Reached := yes;
      else
        Insert u *at the beginning of Waiting;*
    Insert v *into Explored;*
  end
  Remove v *from Waiting;*
end
return(Reached);

Note that both algorithms have an under-specified component, that is, the order in which the successors of the same node are explored. As we will see from the examples, this choice may affect the behavior of the algorithms, especially that of DFS in case a path exists.

Example: Consider the automaton depicted in Figure 3.1 and the following two reachability problems:

- P_1: Is q_5 reachable from q_1 ?
- P_2: Is q_6 reachable from q_1 ?

For P_1, the answer is negative and the two search algorithms end up computing all the states reachable from q_1. In Figure 3.3 we see the behavior of the two algorithms under the following exploration ordering of successors: $q_3 \prec q_2$ for $Succ(q_1)$ and $q_3 \prec q_7 \prec q_4$ for $Succ(q_2)$. The dashed nodes indicate the places where the search stopped, that is, nodes whose successors have not been explored because they have been explored along other paths.

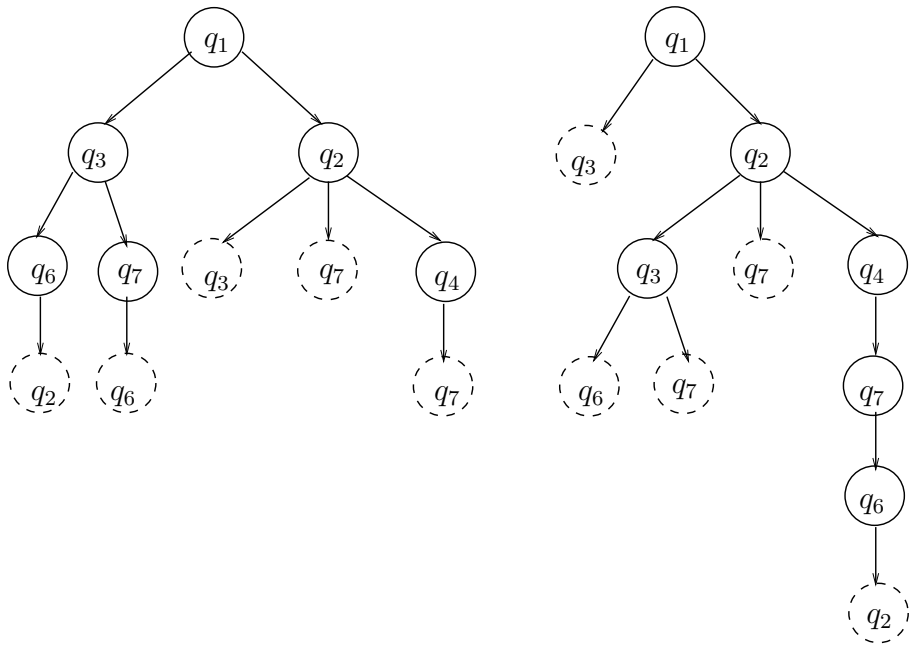
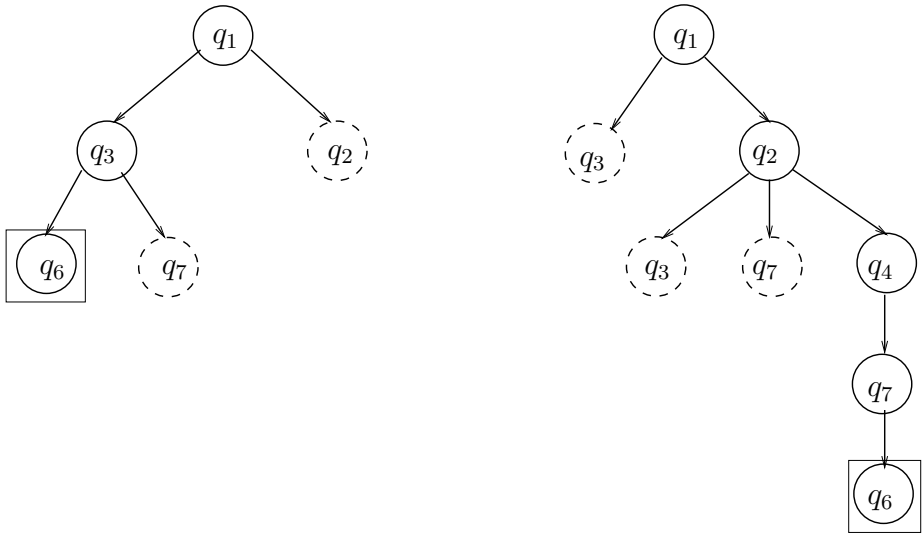


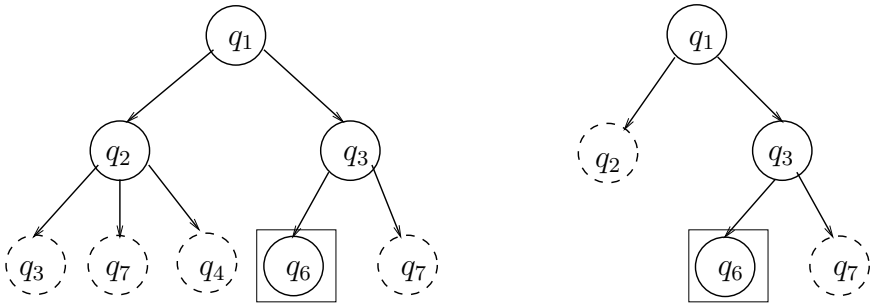
Figure 3.3: The explored search BFS and DFS trees for problem P_1.

The search trees for P_2 under the same successor ordering can be see in Figure 3.4. BFS always finds the shortest (in terms of transitions) path while DFS, under this ordering finds a longer run $q_1 \rightarrow q_2 \rightarrow q_4 \rightarrow q_7 \rightarrow q_6$. The sensitivity of DFS to the ordering of successors is demonstrated in Figure 3.5 where we show the behavior of the algorithm when the successors of q_1 are ordered according to $q_2 \prec q_3$. BFS finds the same shortest path as before (with some less exploration) and DFS now finds this path as well.



Breadth-first search Depth-first search

Figure 3.4: The search trees for P_2 with $q_3 \prec q_2$.



Breadth-first search Depth-first search

Figure 3.5: The search trees for P_2 with $q_2 \prec q_3$.

In certain application domains there might be some useful evaluation function on states that can be used for a more sophisticated search procedure. For example, if the state-space of the automaton consists of Boolean n-tuples, and the dynamics is defined in a certain way, the Hamming distance between a state and q' will give an indication how close we are to our goal. An algorithm that keeps the waiting list ordered according to such a measure is called a *best-first search* algorithm and will be later used extensively for the job-shop problem.

The algorithms described so far explore the automaton in a *forward* direction, starting from q toward q'. Alternatively we could do it backwards, starting from state q' and then computing its predecessors until reaching q or exhausting test set of state from which q' is reached. One way to do a backward search on an automaton $\mathcal{A} = (Q, \delta)$ is to construct an automaton $\mathcal{A}' = (Q, \delta')$ where δ' is the inverse of the transition relation δ. Solving the $q' \to q$ reachability problem in $\mathcal{A}'$ is equivalent to solving the $q \to q'$ reachability problem in $\mathcal{A}$ by backward search.

It is important to mention that the transition graphs of the automata treated in verification are typically very big and are not stored explicitly in the computer memory, but rather generated *on the fly* during the search. The successors of a global state are generated by choosing each time a transition of one of the components (or several components if this transition is common to more than one component) and transforming the global state according to the effect of this transition.

3.2 Shortest Paths

For the automata we will construct for the job-shop problem, the answer to the reachability problem is obvious: in the absence of constraints such as deadlines there will always be a path from the initial state (before any job has started) to the final state (all jobs have terminated). In fact, there will be infinitely many such paths corresponding to the different feasible schedules and the optimization problem is to choose the one with the shortest makespan. This can be done by techniques based on shortest path algorithms, originally applied to weighted automata/graphs.

Weighted automata can be used to associate some cost (energy, time, etc.) with transitions. Or, when the nodes correspond to physical locations, the weight associated with an edge can correspond to the length of the route between its vertices. Formally a weighted automaton is $\mathcal{A} = (Q, \delta, w)$ where $w : \delta \to \mathbb{R}$ is a function that assigns to each transition $(q, q') \in \delta$ a weight $w_{q,q'}$. With every finite run

$$q_0 \xrightarrow{w_{0,1}} q_1 \xrightarrow{w_{1,2}} \dots \xrightarrow{w_{k-1,k}} q_k$$

we associate a cost

$$C = w_{0,1} + w_{1,2} + \ldots + w_{k+1,k}.$$

The shortest path between two states q and q' is the run with the minimal C. From now on we assume that w is always positive and hence the shortest path between any two states has no cycles.

The q-unfolding of a weighted automaton is a tree labeled with pairs in $Q \times \mathbb{R}_+$, starting with $(q, 0)$ and generating for every node (p, r) successors of the form (p', r') such that $(p, p') \in \delta$ and $r' = r' + w_{p,p'}$. Clearly, for any node (p, r) in the tree, r corresponds to the distance from q to p via the corresponding path. A weighted automaton and its unfolding appear in Figures 3.6 and 3.7. Finding the shortest path between q and q' is thus equivalent to finding the node (q', r) with the minimal r in the q-unfolding of the automaton.

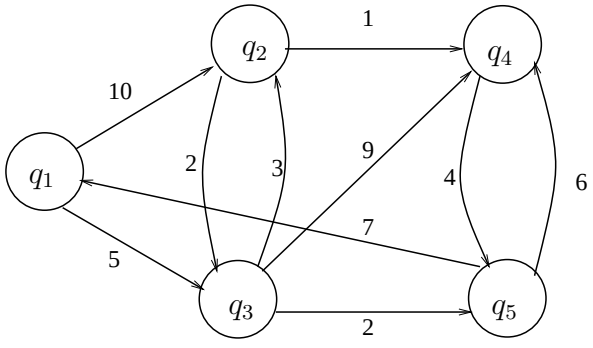


Figure 3.6: A weighted automaton $\mathcal{A}(Q, \delta, w)$

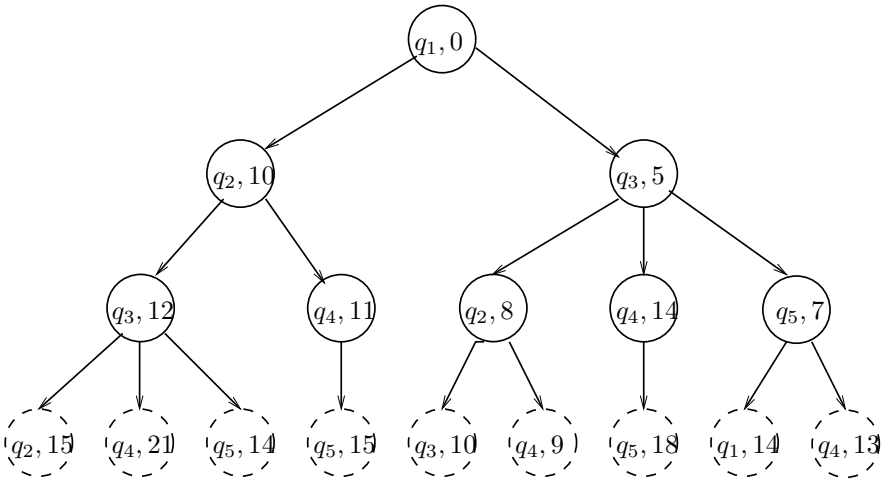


Figure 3.7: The q_1-unfolding of the weighted automaton of Figure 3.6

The BFS algorithm can stop after reaching q' while there exists a shorter path to q' with more transitions. For example, BFS can reach q_4 via the path $q_1 \xrightarrow{10} q_2 \xrightarrow{1} q_4$, Figure 3.8, while there exists a shorter path $q_1 \xrightarrow{5} q_3 \xrightarrow{3} q_2 \xrightarrow{1} q_4$.

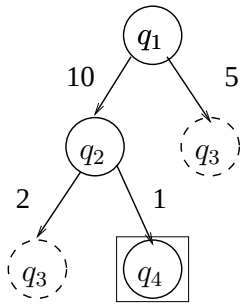


Figure 3.8: BFS algorithm applied to the automata of Figure 3.6

The well-known algorithm due to Dijkstra **??** gives the shortest path from q to any other state for graphs with positive weights. Dijkstra's algorithm can be viewed as iterating with a function $d : Q \to \mathbb{R}_+$ such that at every iteration i, $d^i(p)$ is the length of the shortest path to p having at most i transitions. If p is not reachable within i steps $d^i(p) = \infty$. When the algorithm terminates $d(p)$ holds the length of the shortest path to p.

Algorithm 3 (Dijkstra's algorithm $(\mathcal{A}, q)$**)**

$d^0(q) = 0$;
$\forall p \neq q \; d^0(p) = \infty$;
$i = 0$;
Repeat
begin
$i := i + 1$;
$\forall p \in Q$;
$d^i(p) = min(\{d^{i-1}(p') + W_{pp'}] : p' \in Pre(p)\} \cup \{d^i(p)\})$;
end *Until* $d^i = d^{i+1}$

Chapter 4

Adding Time to Automata

Models based on finite-state automata can express only *qualitative* temporal relations between events in a system. We can say that one events precedes another but we cannot specify in a natural way the quantity of time that separates them. In many application domains such as real-time programming or circuit timing analysis, the correctness of a system depends on the relative speeds of the system and its environment. Hence we need a formalism where we can express *quantitative* timing features of systems such as response time of programs, propagation delays in logical gates or constraints on inter-arrival times of external events. In the context of scheduling problems, we would like to express the duration of a step in a job as a constraint on the time elapsed between its initiation and termination transitions.

Several proposals for extending verification methodology to model quantitative time were proposed in the late 80s and the *timed automaton* model Alur and Dill [ACD93, AD94] turned out to be very useful. This model is rich enough to express all timing problems of practical interest, yet the basic reachability problems for it can be solved algorithmically [HNSY94] using extensions of the graph search algorithms presented in Section Chapter 3.[1]

Timed automata can be viewed as a special class of *hybrid systems*, systems that combine discrete transitions and continuous evolution, of the type usually described by differential equation. This intuition is not so easy to grasp upon first reading. We will present timed automata incrementally, by investigating first the most straightforward approaches for a adding quantitative time to automata.

4.1 Explicit Modeling of Time

Consider two processes P_1 and P_2, with execution times of 3 and 2 time units, respectively. Each process can be in three basic "modes": it can be waiting before starting, it can be active (executing), and it can be finished after having passed enough time in the active mode. The state of the process in the active mode is determined by the amount of time since the process has started. If we fix the time granularity to be *one* time unit we can model the processes using the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ of Figure 4.1. The states of $\mathcal{A}_1$ are $\{\overline{p}_1, 0, 1, 2, 3, \underline{p}_1\}$ where $\overline{p}_1$ is the initial state indicating that the process is still inactive, and $\underline{p}_1$ is the final state. The other states represent the amount of time elapsed in the active mode. The automaton have two types of transition, *immediate* transitions such as "start" and "end", that consume no time, and a special *time-passage* transition "tick" indicating the passage of time unit. The behaviors of the processes are the runs of the automaton consisting of sequences of states and both types of transitions. For example, a behavior where P_1 waits one of one time unit and then starts is

[1] Another discrete formalism for which quantitative timing information can be added are the Petri nets, which are quite popular for modeling manufacturing systems.

captured by the run

$$\overline{p}_1 \xrightarrow{tick} \overline{p}_1 \xrightarrow{start1} 0 \xrightarrow{tick} 1 \xrightarrow{tick} 2 \xrightarrow{tick} 3 \xrightarrow{end1} \underline{p}_1$$

and its corresponding duration is the number of tick transitions.

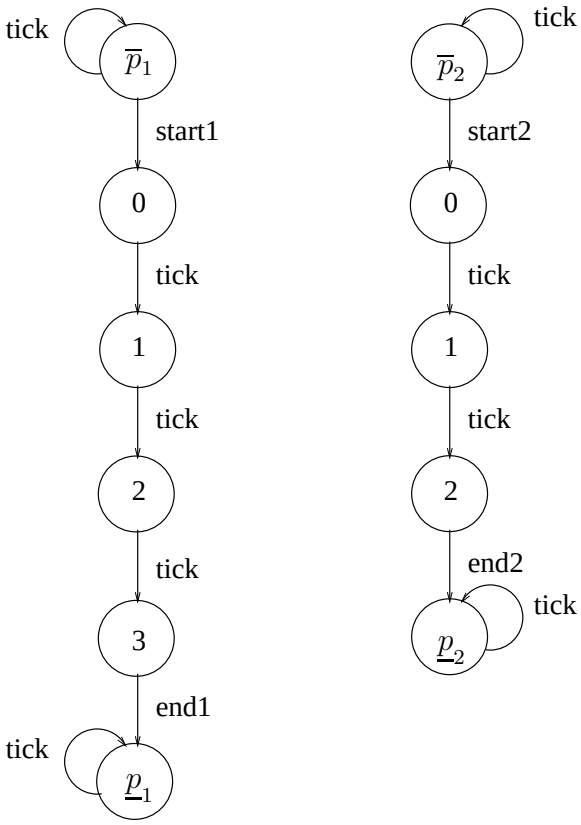


Figure 4.1: The two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ corresponding to P_1 and P_2

When two or more automata run in parallel, each automaton can take its immediate transitions independently but the "tick" transitions are synchronized: if one process takes such a transition, all the the others need to take it as well. The effect of the tick transition on any active process in a state i is is to move it to state $i+1$. The automaton $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$ is shown in Figure 4.2. It starts in an initial state $(\underline{p}_1, \underline{p}_2)$ where the two processes are waiting. Each of the processes can start executing after any number of ticks and all the possible behaviors of the system correspond to runs of $\mathcal{A}$. For example, the behavior where both processes wait 2 time units and then start at the same time[2] is captured by the run:

$$(\overline{p}_1, \overline{p}_2) \xrightarrow{tick} (\overline{p}_1, \overline{p}_2) \xrightarrow{tick} (\overline{p}_1, \overline{p}_2) \xrightarrow{start1} (0, \overline{p}_2) \xrightarrow{start2} (0,0) \xrightarrow{tick}$$
$$(1,1) \xrightarrow{tick} (2,2) \xrightarrow{end2} (2, \underline{p}_2) \xrightarrow{tick} (3, \underline{p}_2) \xrightarrow{end1} (\underline{p}_1, \underline{p}_2)$$

A behavior where P_1 starts immediately and P_2 starts 2 time units later is represented by the

[2]In fact, behaviors where several independent immediate transitions occur at the same time can usually be represented by more than one run, each run corresponding to a different *interleaving* of concurrent transitions. For example, the run fragment $(\overline{p}_1, \overline{p}_2) \xrightarrow{start1} (0, \overline{p}_2) \xrightarrow{start2} (0,0)$ can be replaced by $(\overline{p}_1, \overline{p}_2) \xrightarrow{start2} (\overline{p}_1, 0) \xrightarrow{start1} (0,0)$.

run

$$(\overline{p}_1, \overline{p}_2) \xrightarrow{start1} (0, \overline{p}_2) \xrightarrow{tick} (1, \overline{p}_2) \xrightarrow{tick} (2, \overline{p}_2) \xrightarrow{start2}$$
$$(2, 0) \xrightarrow{tick} (3, 1) \xrightarrow{end1} (\underline{p}_1, 1) \xrightarrow{tick} (\underline{p}_1, 2) \xrightarrow{end2} (\underline{p}_1, \underline{p}_2)$$

A behavior where P_2 starts immediately and P_1 starts 1 unit after P_2 terminates is represented by the run

$$(\overline{p}_1, \overline{p}_2) \xrightarrow{start2} (\overline{p}_1, 0) \xrightarrow{tick} (\overline{p}_1, 1) \xrightarrow{tick} (\overline{p}_1, 2) \xrightarrow{end2} (\overline{p}_1, \underline{p}_2) \xrightarrow{tick}$$
$$(\overline{p}_1, \underline{p}_2) \xrightarrow{start1} (0, \underline{p}_2) \xrightarrow{tick} (1, \underline{p}_2) \xrightarrow{tick} (2, \underline{p}_2) \xrightarrow{tick} (3, \underline{p}_2) \xrightarrow{end1} (\underline{p}_1, \underline{p}_2)$$

The duration of a run, the time from the initial state to $(\underline{p}_1, \underline{p}_1)$, is just the number of ticks in the run (5, 4 and 6 units, respectively).

Since each processes can choose to perform its start transition independently, many possible combination of clock values are possible, each reflecting a different choice in the past. This can lead to a state explosion as the number of processes grow. In addition, each refinement of the time scale (e.g. letting events occur in time instants that are multiples of 1/2) will increase the number of states in each automaton. The advantage of this representation is, however, that it allows us to stay within the familiar framework of automata and to apply standard reachability and shortest-path algorithms to timed systems, by assigning weight 1 to tick transitions and 0 to immediate transitions

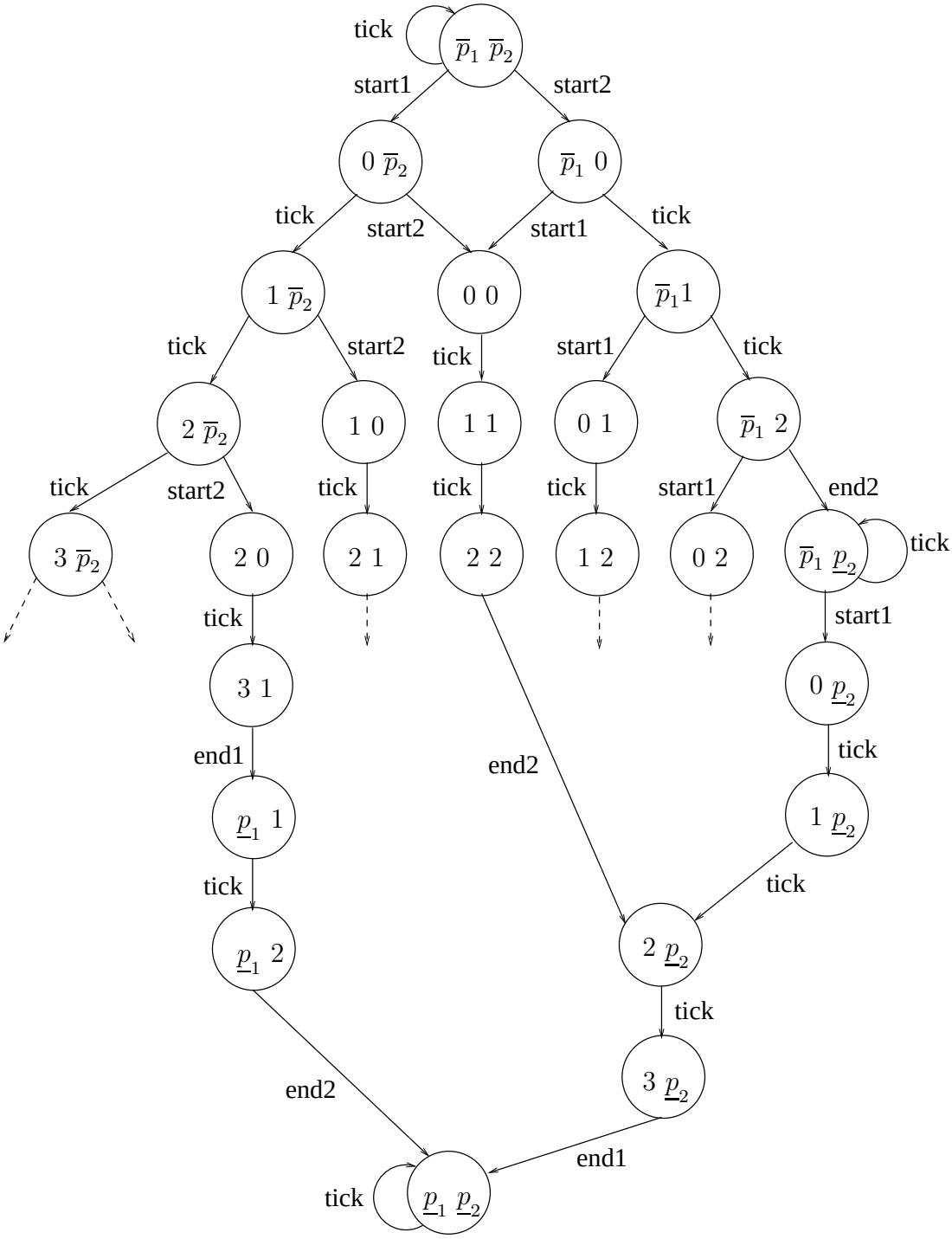


Figure 4.2: The global automaton of $\mathcal{A} = \mathcal{A}_1||\mathcal{A}_2$.

4.2 Clock Variables

A more compact representation of such automata can be achieved by using special *auxiliary variables* to represent time passage instead of encoding elapsed time explicitly inside states. These are clock variables or counters that are reset to zero when an active state is entered,

incremented by one with every tick and their value is tested in transitions that leave active states. Figure 4.3 shows how this is done for the automata of Figure 4.1, by adding clock variables X_1 and X_2. A state (or configuration) of an augmented automaton is a pair of the form $(q, \mathbf{v})$ where q is an explicit state and $\mathbf{v}$ is a value for the variable(s). Such clock variables can range over the non-negative integers with the special value $\perp$ indicating that the clock is not active in a state. A run of $\mathcal{A}'_1$ will look like

$$(\overline{p}_1, \perp) \xrightarrow{tick} (\overline{p}_1, \perp) \xrightarrow{atart1} (p_1, 0) \xrightarrow{tick} (p_1, 1) \xrightarrow{tick} (p_1, 2) \xrightarrow{tick} (p_1, 3) \xrightarrow{end1} (\underline{p}_1, \perp)$$

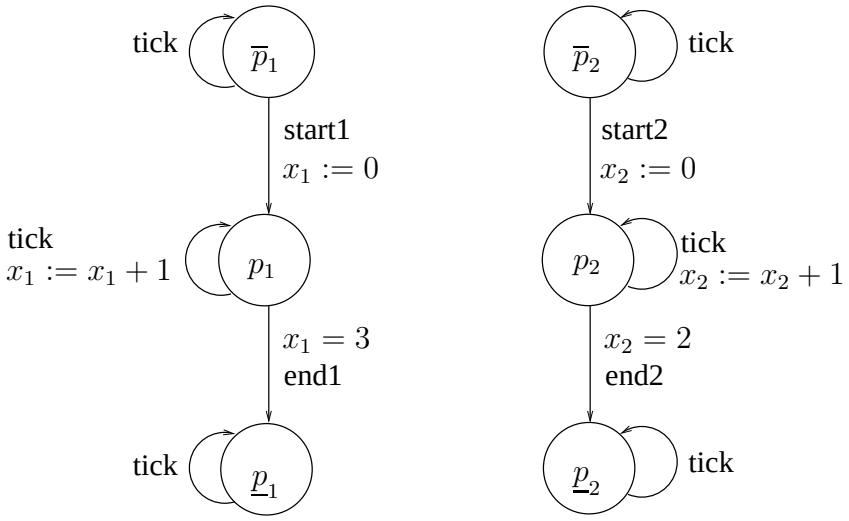


Figure 4.3: The automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$: using clocks variables.

Note that the difference between the two approaches is purely syntactic. If we expand the automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$ by adding clock values to the states we obtain automata isomorphic to $\mathcal{A}_1$ and $\mathcal{A}_2$ (see Figure 4.4).

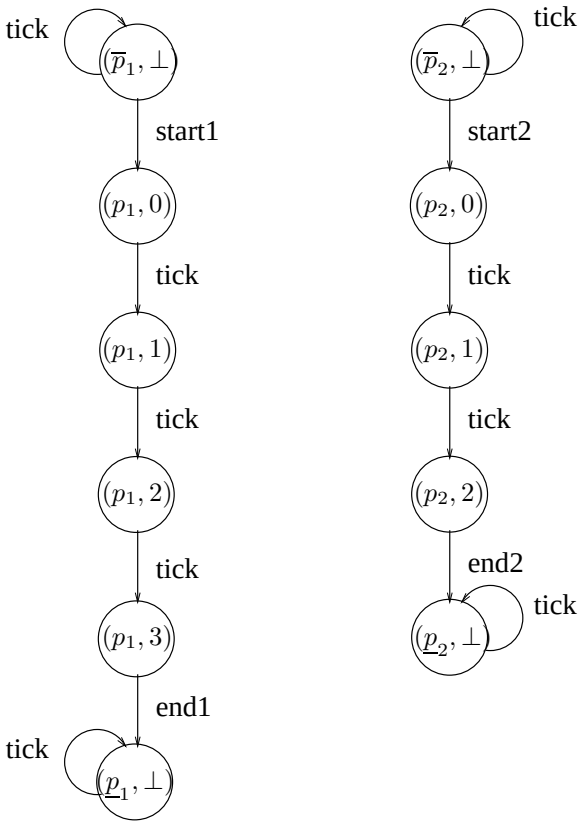


Figure 4.4: The automata $\mathcal{A}'_1$ and $\mathcal{A}'_2$ expanded with explicit representation of clock values in the state.

When we compose $\mathcal{A}'_1$ and $\mathcal{A}'_2$ we obtain the global automaton $\mathcal{A}'$ of Figure 4.5 which looks simpler than $\mathcal{A}$ of Figure 4.2. This simplicity of the transition graph is, however, misleading. Consider for example state (p_1, p_2) where both processes are active. There are two transitions leaving this state and they are guarded by conditions $X_1 = 3$ and $X_2 = 2$, respectively. The state itself does not tell us whether each of the transitions can be taken as this depends on the values of the clocks which, in turn, depends on the previous history (when the clocks were reset to zero). In the worst case, reachability algorithms for $\mathcal{A}'$ might need to expand it into $\mathcal{A}$. Nevertheless, although modeling with clock variables does not change the worst-case complexity of the reachability problem, it allows us to use *symbolic methods* and work with an arbitrary time granularity.

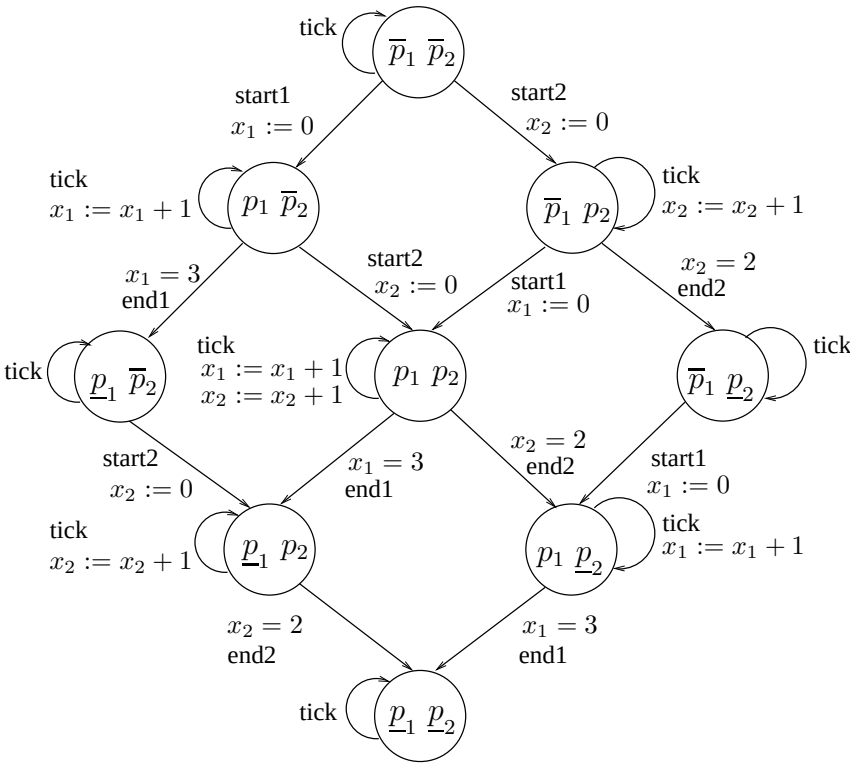


Figure 4.5: The global automaton $\mathcal{A}' = \mathcal{A}'_1 || \mathcal{A}'_2$.

Symbolic or implicit representation methods can be applied to variables that belong to some mathematical domain such as integers. Instead of representing a set of states explicitly (e.g. in a table) it can be represented by a formula. Suppose, for example, two processes with durations d_1 and d_2, respectively such that $d_1 < d_2$, that enter their respective active states 2 time units one after the other. The set of reachable clock values has the form

$$\{(2,0),(3,1),(4,2),\ldots(d_1,d_1-2)\}$$

and its size depends on d_1. However the formula $X_1 - X_2 = 2 \wedge X_1 \leq d_1$ characterizes this set, and its size does not depends on d_1. In fact, it can characterize the reachable states even if we do not assume any fixed time granularity and work with dense time. This way we may allow events happen anywhere on the real-time axis and view the clocks differently, as continuous variables that evolve with derivative 1 inside active states. These are timed automata, see Figures 4.6 and 4.7, which can be seen as the limit of a process that makes the time steps associated with tick transitions infinitesimal.

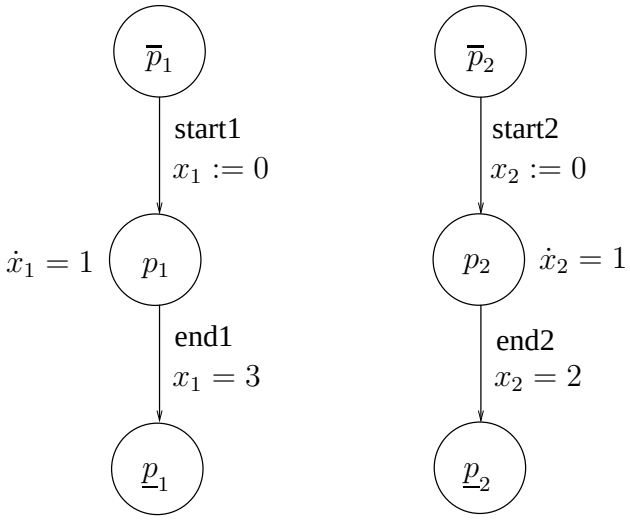


Figure 4.6: Two Timed Automata

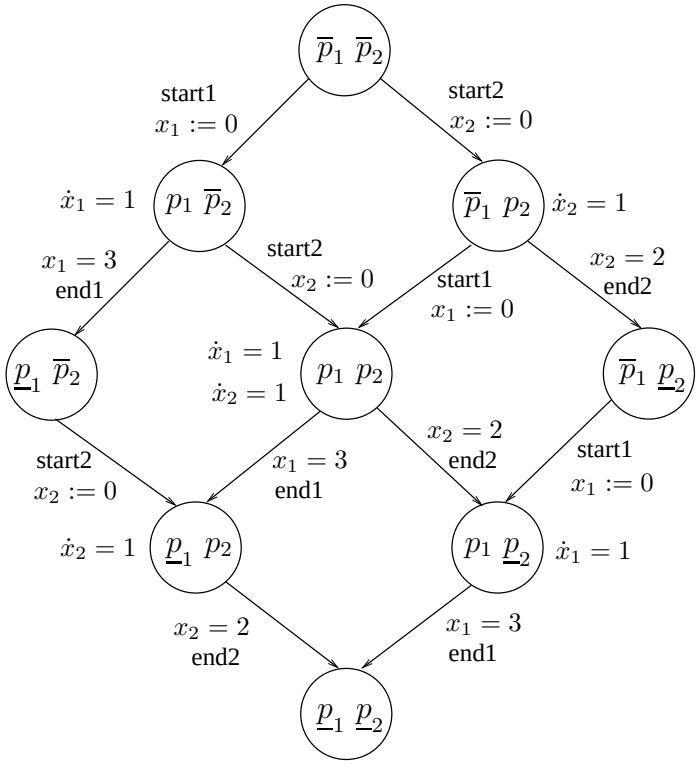


Figure 4.7:

4.3 Basics of Timed Automata

Definition 2 (Timed Automata) *A* timed automaton *is a tuple* $\mathcal{A} = (Q, C, s, f, \Delta)$ *where* Q *is a finite set of states,* C *is a finite set of clocks, and* Δ *is a transition relation consisting of*

elements of the form (q, ϕ, ρ, q') *where* q *and* q' *are states,* $\rho \subseteq C$ *and* ϕ *(the transition guard) is a boolean combination of formulae of the form* $(c \in I)$ *for some clock* c *and some integer-bounded interval* I*. States* s *and* f *are the initial and final states, respectively.*

Definition 3 (Clock Valuation) *A* clock valuation *is a function* $\mathbf{v} : C \to \mathbb{R}_+ \cup \{0\}$*, or equivalently a* $|C|$*-dimensional vector over* $\mathbb{R}_+$*. We denote the set of all clock valuations by* $\mathcal{H}$*. A configuration of the automaton is hence a pair* $(q, \mathbf{v}) \in Q \times \mathcal{H}$ *consisting of a discrete state (sometimes called "location") and a clock valuation. Every subset* $\rho \subseteq C$ *induces a reset function* $\mathrm{Reset}_\rho : \mathcal{H} \to \mathcal{H}$ *defined for every clock valuation* $\mathbf{v}$ *and every clock variable* $c \in C$ *as*

$$\mathrm{Reset}_\rho \, \mathbf{v}(c) = \begin{cases} 0 & \text{if} \quad c \in \rho \\ \mathbf{v}(c) & \text{if} \quad c \notin \rho \end{cases}$$

That is, Reset_ρ *resets to zero all the clocks in* ρ *and leaves the other clocks unchanged. We use* $\mathbf{1}$ *to denote the unit vector* $(1, \ldots, 1)$ *and* $\mathbf{0}$ *for the zero vector.*

Definition 4 (Steps and Runs) *A* step *of the automaton is one of the following:*

- *A discrete step:* $(q, \mathbf{v}) \xrightarrow{0} (q', \mathbf{v}')$*, where there exists* $\delta = (q, \phi, \rho, q') \in \Delta$*, such that* $\mathbf{v}$ *satisfies* ϕ *and* $\mathbf{v}' = \mathrm{Reset}_\rho(\mathbf{v})$*.*
- *A time step:* $(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1})$*,* $t \in \mathbb{R}_+$*.*

A run *of the automaton starting from a configuration* $(q_0, \mathbf{v}_0)$ *is a finite sequence of steps*

$$\xi : \quad (q_0, \mathbf{v}_0) \xrightarrow{t_1} (q_1, \mathbf{v}_1) \xrightarrow{t_2} \cdots \xrightarrow{t_n} (q_n, \mathbf{v}_n).$$

The logical length *of such a run is* n *and its* metric length *is* $t_1 + t_2 + \cdots + t_n$*.*

It is sometimes useful to augment a time automaton $\mathcal{A}$ with an additional clock t which is active in every state and never reset to zero. We call the obtained automaton $\mathcal{A}'$ the *extended automaton* of $\mathcal{A}$, and its runs are called *extended runs* of $\mathcal{A}'$. Since t always represents absolute time, $(q, \mathbf{v}, t)$ is reachable in $\mathcal{A}'$ iff $(q, \mathbf{v})$ is reachable in $\mathcal{A}$ at time t.

Note that we omit transition labels, such as "start" or "end", used in the previous section from the runs because we will not need them in the sequel, and we use their duration, 0, instead. Although in the formal definition all clocks evolve uniformly with derivative 1 in all states, there are states were certain clock values are not important because in all paths starting from those states they are not tested before being reset to zero (for example, clock X_1 in state $(\overline{p}_1, p_2)$). We say that a clock is *inactive* in such a state and instead of writing down its value we use the symbol $\perp$. The following run of the automaton $\mathcal{A}''$ of Figure 4.7 corresponds to the case where P_1 started after 0.23 time units and P_2 started 2.1 time units later:

$$(\overline{p}_1, \overline{p}_2, \perp, \perp) \xrightarrow{0.23} (p_1, \overline{p}_2, 0, \perp) \xrightarrow{2.1} (p_1, \overline{p}_2, 2.1, \perp) \xrightarrow{0} (p_1, p_2, 2.1, 0) \xrightarrow{0.9}$$

$$(p_1, p_2, 3, 0.9) \xrightarrow{0} (p_1, \underline{p}_2, \perp, 0.9) \xrightarrow{1.1} (p_1, \underline{p}_2, \perp, 2) \xrightarrow{0} (\underline{p}_1, \underline{p}_2, \perp, \perp)$$

Note also that the definition of a run allows to "split" time steps, for example, the step $(p_1, \overline{p}_2, 0, \perp) \xrightarrow{2.1} (p_1, \overline{p}_2, 2.1, \perp)$ can be written as $(p_1, \overline{p}_2, 0, \perp) \xrightarrow{0.6} (p_1, \overline{p}_2, 0.6, \perp) \xrightarrow{0.7} (p_1, \overline{p}_2, 1.3, \perp) \xrightarrow{0.8} (p_1, \overline{p}_2, 2.1, \perp)$.

One of the most useful features of timed automata is their ability to epxress and analyze system with temporal uncertainty. In the automaton of Figure 4.8 we see that the transition

from q_1 to q_2 can happen when the value of X is anywhere in the interval $[1, 3]$. A verification algorithm should explore all these infinitely-many runs that correspond to this choice, and the major result on timed automata [AD94, HNSY94, ACD93] is that although the state space is infinite, reachability and other verification problems for timed automaton are solvable.

The basic idea for reachability computation for timed automata is the following:

1. Sets of reachable configurations are stored as unions of *symbolic states* of the form (q, Z) where q is a discrete state and Z is a subset of the clock space $\mathcal{H}$.
2. Computation of successors of a symbolic state is done in two phases. First all the time successors of (q, Z) are computed leading to (q, Z') where Z' consists of all clock valuation reachable from Z by letting time progress by any amount. Then Z' is intersected with the transition guard of each transition to determine the configuration where the transition can take place and then the reset operations is applied to those configurations.

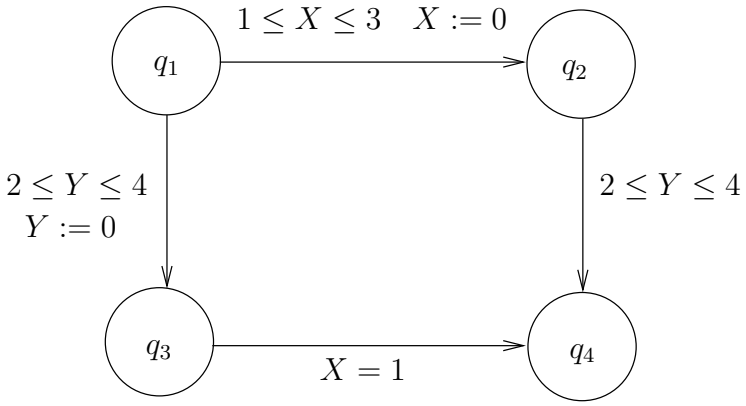


Figure 4.8: A timed automaton.

Consider the automaton of Figure 4.8. Starting from the symbolic state $(q_1, X = Y = 0)$, letting time pass we reach the symbolic state $(q_1, X = Y \geq 0)$. The intersection with the guard of the transition to q_2 gives $(q_1, 1 \leq X = Y \leq 3)$ and the resetting of X leads us finally to $(q_2, X = 0 \wedge 1 \leq Y \leq 3)$ from where we restart with time passage and so on. The whole computation for that automaton appears in Figure 4.9. The fundamental result concerning timed automata is that the sets of clock valuations in symbolic states always belong to a special class of polyhedra called *zones* which is finite for any given automaton.

Definition 5 (Zones and Symbolic States) *A* zone *is a subset of $\mathcal{H}$ consisting of points satisfying a conjunction of inequalities of the form $c_i - c_j \geq d$ or $c_i \geq d$. A* symbolic state *is a pair (q, Z) where q is a discrete state and Z is a zone. It denotes the set of configurations $\{(q, z) : z \in Z\}$.*

Definition 6 (Successors) *Let $\mathcal{A} = (Q, C, s, f, \Delta)$ be a timed automaton and let (q, Z) be a symbolic state.*

- *The* time successor *of (q, Z) is the set of configurations which are reachable from (q, Z) by letting time progress:*

$$Post^t(q, Z) = \{(q, z + r\mathbf{1}) : z \in Z, r \geq 0\}.$$

We say that (q, Z) is time-closed *if $(q, Z) = Post^t(q, Z)$.*

- *The* δ-transition successor *of* (q, Z) *is the set of configurations reachable from* (q, Z) *by taking the transition* $\delta = (q, \phi, \rho, q') \in \Delta$*:*

$$Post^{\delta}(q, Z) = \{(q', \text{Reset}_{\rho}(z)) : z \in Z \cap \phi\}.$$

- *The* δ-successor *of a time-closed symbolic state* (q, Z) *is the set of configurations reachable by a* δ*-transition followed by passage of time:*

$$Succ^{\delta}(q, Z) = Post^{t}(Post^{\delta}(q, Z)).$$

- *The* successors *of* (q, Z) *is the set of all its* δ*-successors:*

$$Succ(q, Z) = \bigcup_{\delta \in \Delta} (Succ^{\delta}(q, Z)).$$

Equipped with these operations (which transform zones into zones) we can solve reachability problems for timed automata using graph search algorithms that work on the "simulation graph", a graph whose nodes are symbolic states connected by the successor relation. This approach for verifying timed automata was first proposed in [HNSY94] and implemented in the tool KRONOS [Y97]. As we will see in the next chapter, the timed automata for modeling job-shop problems have a special structure and in particular they are acyclic. The following generic algorithm computes all the reachable configuration for such automata starting from configuration $(s, \mathbf{0})$.

Algorithm 4 (Forward Reachability for Acyclic Timed Automata)
Waiting:=$\{Post^{t}\{(s, \mathbf{0})\}\}$*;*
while *Waiting* $\neq \emptyset$ **do**
Pick $(q, Z) \in$ *Waiting;*
For every $(q', Z') \in Succ(q, Z)$*;*
Insert (q', Z') *into Waiting;*
Remove (q, Z) *from Waiting;*
end

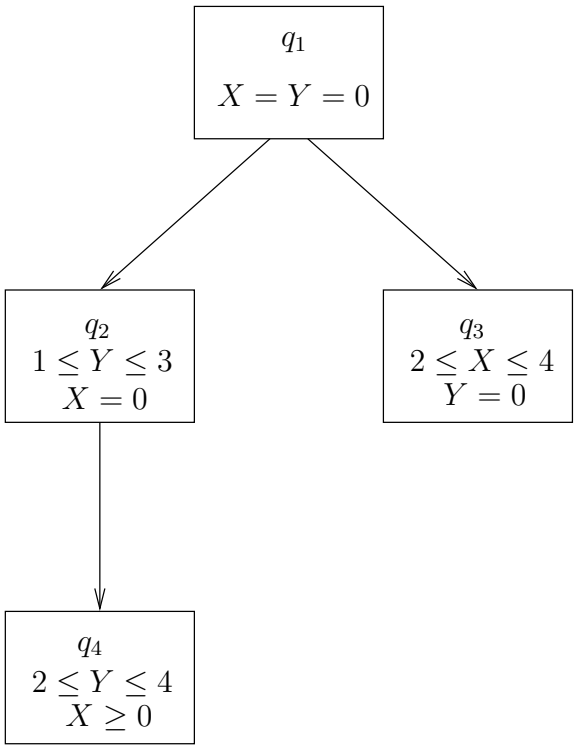


Figure 4.9: Forward reachability computed for the automaton of Figure 4.8

Since the automaton is acyclic the algorithm will terminate even if we do not keep a list of explored states. However, for performance reasons such tests are important as we will see later, especially when there are many paths that lead to the same discrete state.

As in untimed automata, reachable sets can be computed backwards by computing the predecessors of a symbolic state. The definitions are similar to that of successors, except for the fact the we need to compute inverse images of reset functions.

Definition 7 (Predecessors) *Let $\mathcal{A} = (Q, C, s, f, \Delta)$ be a timed automaton and let (q, Z) be a symbolic state.*

- *The* time predecessors *of (q, Z) is the set of configurations from which (q, Z) can be reached by letting time progress:*

$$Pre^t(q, Z) = \{(q, \mathbf{v}) : \mathbf{v} + r\mathbf{1} \in Z, r \geq 0\}.$$

  We say that (q, Z) is time-closed *if $(q, Z) = Pre^t(q, Z)$.*

- *The δ-*transition predecessor *of (q, Z) is the set of configurations from which (q, Z) is reachable by taking the transition $\delta = (q', \phi, \rho, q) \in \Delta$:*

$$Pre^\delta(q, Z) = \{(q', \mathbf{v}') : \mathbf{v}' \in \text{Reset}_\rho^{-1}(Z) \cap \phi\}.$$

- *The* predecessors *of (q, Z) is the set of all configuration from which (q, Z) is reachable by any transition δ followed by passage of time:*

$$Pre(q, Z) = \bigcup_{\delta \in \Delta} Pre^t(Pre^\delta(q, Z)).$$

The following algorithm computes the set of states from which a final state f is reachable.

Algorithm 5 (Backward Reachability for Acyclic Timed Automata)
Waiting:=$\{(f, \mathcal{H})\}$*;*
while *Waiting* $\neq \emptyset$ **do**
Pick $(q, Z) \in$ *Waiting;*
For every $(q', Z') \in Pre(q, Z)$*;*
Insert (q', Z') *into Waiting;*
Remove (q, Z) *from Waiting;*
end

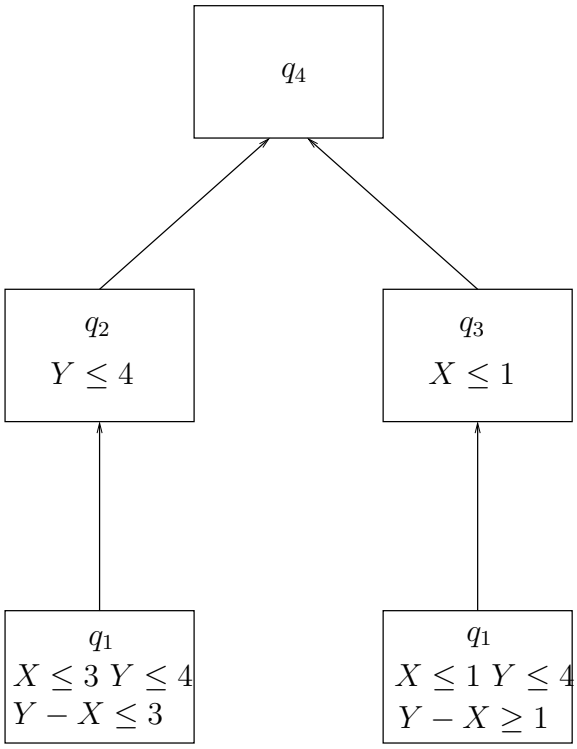


Figure 4.10: Backward reachability computed for the automaton of Figure 4.8

Part II

Contribution

Chapter 5

Deterministic Job Shop Scheduling

In this chapter we model the job-shop scheduling problem using a special class of acyclic timed automata. Finding an optimal schedule corresponds, then, to finding a shortest (in terms of elapsed time) path in the automaton. This representation provides new techniques for solving the optimization problem. We present algorithms and heuristics for finding shortest paths in timed automata and test their implementation on numerous benchmark examples.

5.1 Formal definitions

Definition 8 (Job-Shop Specification)
Let M be a finite set of resources (machines). A job specification over a set M of resources is a triple $J = (k, \mu, d)$ where $k \in \mathbb{N}$ is the number of steps in J, $\mu : \{1..k\} \to M$ indicates which resource is used at each step, and $d : \{1..k\} \to \mathbb{N}$ specifies the length of each step. A job-shop specification is a set $\mathcal{J} = \{J^1, \ldots, J^n\}$ of jobs with $J^i = (k^i, \mu^i, d^i)$.

We make the assumption that each machine is used exactly once by every job. This assumption simplifies the presentation but still maintains the inherent complexity. We denote $\mathbb{R}_+$ by T, abuse $\mathcal{J}$ for $\{1, \ldots, n\}$ and let $K = \{1, \ldots, k\}$.

Definition 9 (Feasible Schedules)
A feasible schedule for a job-shop specification $\mathcal{J} = \{J^1, \ldots, J^n\}$ is a relation $S \subseteq \mathcal{J} \times K \times T$ so that $(i, j, t) \in S$ indicates that job J^i is busy doing its j^{th} step at time t and, hence, occupies machine $\mu^i(j)$. A feasible schedule should satisfy the following conditions:

1. Ordering*: if $(i, j, t) \in S$ and $(i, j', t') \in S$ then $j < j'$ implies $t < t'$ (steps of the same job are executed in order).*
2. Covering and Non-Preemption*: For every $i \in \mathcal{J}$ and $j \in K$, the set $\{t : (i, j, t) \in S\}$ is a non-empty set of the form $[r, r + d]$ for some $r \in T$ and $d = d^i(j)$ (every step is executed continuously until completion).*
3. Mutual Exclusion*: For every $i, i' \in \mathcal{J}$, $j, j' \in K$ and $t \in T$, if $(i, j, t) \in S$ and $(i', j', t) \in S$ then $\mu^i(j) \neq \mu^{i'}(j')$ (two steps of different jobs which execute at the same time do not use the same machine).*

The start time time of step j in job i is

$$s(i, j) = \min_{(i,j,t) \in S} t$$

The *length* of a schedule is the maximal t over all $(i, j, t) \in S$.

The *optimal schedule* of a job-shop specification $\mathcal{J}$ is a feasible schedule with the shortest length.

From the relational definition of schedules one can derive the following commonly-used definition, namely

1. The *machine allocation function* $\alpha : M \times T \to \mathcal{J}$ stating which job occupies a machine at any time, defined as $\alpha(m, t) = i$ if $(i, j, t) \in S$ and $\mu^i(j) = m$.
2. The *task progress function* $\beta : \mathcal{J} \times T \to M$ stating what machine is used by a job is at a given time, defined as $\beta(i, t) = m$ if $(i, j, t) \in S$ and $\mu^i(j) = m$.

The machine allocation and task progress function are partial, because a machine or a job might be idle at certain times.

A machine m is *idle* at t, $\alpha(m, t) = \perp$, if

$$\forall i, j \text{ s.t. } \mu^i(j) = m \ (i, j, t) \notin S.$$

A job i is *idle* at t, $\beta(i, t) = \perp$, if

$$\forall j \text{ s.t. } \mu^i(j) = m \ (i, j, t) \notin S.$$

A step j in a job i is *enabled* at t if

$$s(i, j-1) + d_i(j-1) < t < s(i, j)$$

and

$$\alpha(m, t) = \perp.$$

As an example consider $M = \{m_1, m_2, m_3\}$ and two jobs

$$J^1 = (m_3, 2), (m_2, 2), (m_1, 4) \text{ and } J^2 = (m_2, 3), (m_3, 1)$$

Two schedules S_1 and S_2 are depicted in Figure 5.1 in both machine allocation and task progress forms. The length of S_2 is 8 and it is the optimal schedule.

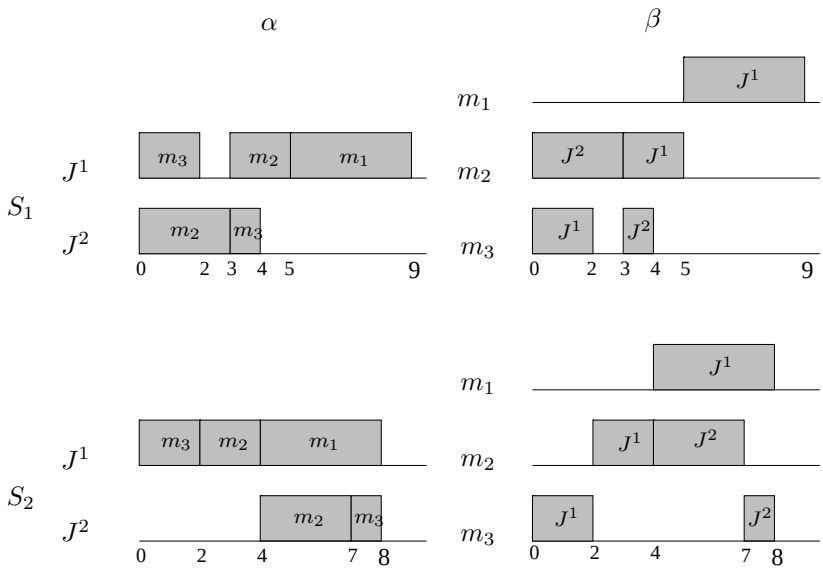


Figure 5.1: Two feasible schedules S_1 and S_2 visualized as the machine allocation function α and the task progress function β.

Note that a job can be idle at time t even if its precedence constraints are satisfied and the machine needed at this time is available. As one can see in schedule S_2 of Figure 5.1, machine m_2 is available at time $t = 0$ whereas J^2 does not use it and remains idle until time $t = 4$. If we execute the steps of J^2 as soon as they are enabled we obtain the longer schedule S_1.

The ability to achieve the optimum by waiting instead of starting immediately increases the set of feasible solutions that need to be explored and is the major source of the complexity of scheduling.

5.2 Modeling with Timed Automata

5.2.1 Modeling Jobs

We construct for every job $J = (k, \mu, d)$ a timed automaton with one clock c and $2k + 1$ states. For every step j such that $\mu(j) = m$ there will be two states in the timed automata, a state $\overline{m}$ which indicates that the job is waiting to start the step and a state m indicating that the job is executing the step. The initial state is the state $\overline{\mu}(1)$ where the job J has not started yet, and the final state is the state f where the job has terminated all its steps. The clock c is *inactive* at states $\overline{m}$ and upon entering m it is reset to zero without being tested. The automaton can leave the state m only after time $d(j)$ has elapsed, this is done while testing if the clock c is larger or equal to $d(j)$.

Let $\overline{M} = \{\overline{m} : m \in M\}$ and let $\overline{\mu} : K \to \overline{M}$ be an auxiliary function such that $\overline{\mu}(j) = \overline{m}$ whenever $\mu(j) = m$.

Definition 10 (Timed Automaton for a Job)
Let $J = (k, \mu, d)$ be job. Its associated timed automaton is $\mathcal{A} = (Q, \{c\}, \Delta, s, f)$ with $Q = P \cup \overline{P} \cup \{f\}$ where $P = \{\mu(1), \ldots \mu(k)\}$, and $\overline{P} = \{\overline{\mu}(1), \ldots, \overline{\mu}(n)\}$. The initial state is $\overline{\mu}(1)$.

The transition relation Δ *consists of the following tuples*

$$\begin{array}{ll} (\overline{\mu}(j), true, \{c\}, \mu(j)) & j = 1..k \\ (\mu(j), c = d(j), \emptyset, \overline{\mu}(j+1)) & j = 1..k-1 \\ (\mu(k), c = d(k), \emptyset, f) & \end{array}$$

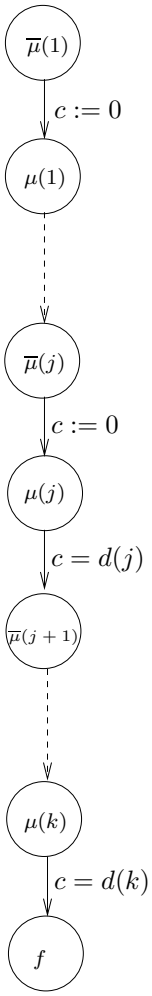


Figure 5.2: The automaton corresponding to the job $J = (k, \mu, d)$.

5.2.2 Modeling Job Shop Specifications

To construct the timed automaton for the whole job shop specification we need to compose the automata for the individual tasks. The composition is rather standard, the only particular feature is the enforcement of mutual exclusion constraints by forbidding *conflicting states*, that is, global states in which two or more automata are in a state corresponding to the same resource m.

Definition 11 (Conflicting states)
An n*-tuple* $q = (q^1, \ldots, q^n) \in Q^n$ *is said to be* conflicting *if it contains two components* q^a *and* q^b *such that* $q^a = q^b = m \in M$*.*

Let be $\mathcal{J} = \{J^1, \ldots, J^n\}$ a job shop specification, and let be $\mathcal{A}^i = (Q^i, \{c_i\}, \Delta^i, s^i, f^i)$ the timed automaton for job J^i. We compose these automata and obtain a time automaton $\mathcal{A} = (Q, C, \Delta, s, f)$ with n clocks. The states of the composition is the Cartesian product of the states of the individual automata, excluding conflicting states.

Definition 12 (Mutual Exclusion Composition)
Let $\mathcal{J} = \{J^1, \ldots, J^n\}$ be a job-shop specification and let $\mathcal{A}^i = (Q^i, C^i, \Delta^i, s^i, f^i)$ be the automaton corresponding to each J^i. Their mutual exclusion composition is the automaton $\mathcal{A} = (Q, C, \Delta, s, f)$ such that Q is the restriction of $Q^1 \times \ldots Q^n$ to non-conflicting states, $C = C^1 \cup \ldots \cup C^n$, $s = (s^1, \ldots, s^n)$, $f = (f^1, \ldots, f^n)$ and the transition relation Δ contains all the tuples of the form

$$((q^1, \ldots, q^a, \ldots, q^n), \phi, \rho, (q^1, \ldots, p^a, \ldots, q^n))$$

such that $(q^a, \phi, \rho, p^a) \in \Delta^a$ for some a and both $(q^1, \ldots, q^a, \ldots, q^n)$ and $(q^1, \ldots, p^a, \ldots, q^n)$ are non-conflicting.

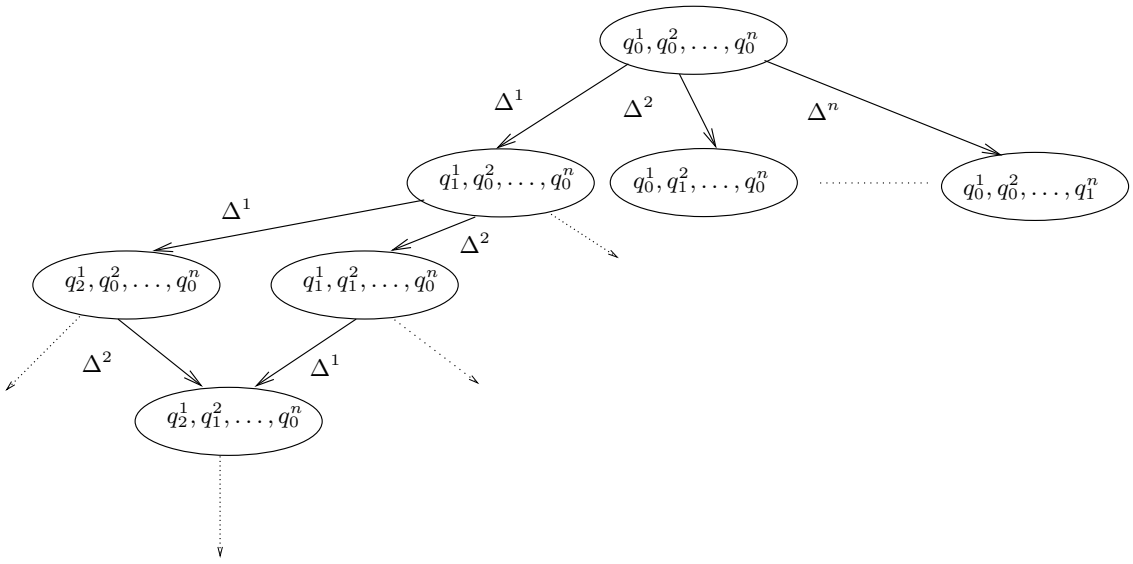


Figure 5.3: The timed automaton corresponding to the job shop specification $\mathcal{J} = \{J^1, J^2, \ldots, J^n\}$

As can be seen from the definition and Figure 5.3, each discrete transition in $\mathcal{A}$ corresponds to a transition in *one* automaton. This is called the *interleaving semantics* and it is used only for technical convenience. If in a schedule two automata make transitions δ^1, δ^2 at the same time instant, there will be two corresponding run fragments $q \xrightarrow{\delta 1} q' \xrightarrow{\delta 2} p$ and $q \xrightarrow{\delta 2} q'' \xrightarrow{\delta 1} p$ in the automaton.

As an example consider $M = \{m_1, m_2\}$ and two jobs

$$J^1 = (m_1, 4), (m_2, 5) \text{ and } J^2 = (m_1, 3)$$

The corresponding automata for the two jobs are depicted in Figure 6.4 and there composition in Figure 8.5.

The job-shop timed automaton is acyclic and "diamond-shaped" with an initial state in which no job has started and a final state where all jobs have terminated. A run of this automaton is called *complete* if it starts at s and terminates at f.

All transitions in the global timed automaton indicate either a component moving from an active to an inactive state (these are guarded by conditions of the form $c_i = d$), or a component moving into an active state (these are labeled by resets $c_i := 0$ and are called *start$_i$ transitions*).

Two transitions outgoing from the same state might represent a choice of the scheduler, for example, the two transitions outgoing from the initial state represent the decision to whom to give first the resource m_1, either to J^1 and move to $(m_1, \overline{m}_1)$, or to J^2 and move to $(\overline{m}_1, m_1)$. The scheduler can also decide to start a job or let it idle, as we can see in the two transitions outgoing from the state $(m_2, \overline{m}_1)$. The scheduler either starts job J^2 on machine m_1 or let time pass until clock c_1 satisfies the guard, and move to $(f, \overline{m}_1)$. On the other hand some duplication of paths are just artifacts due to interleaving, for example, the two paths outgoing from $(\overline{m}_2, \overline{m}_1)$ to (m_2, m_1) are practically equivalent. Recall that in a timed automaton, the transition graph might be misleading, because two or more transitions entering *the same* discrete state, e.g. transitions to (m_2, f) might enter it with different clock valuations, and hence lead to different continuations.

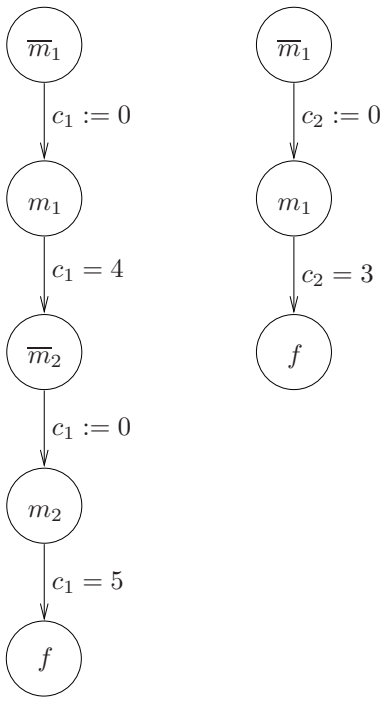


Figure 5.4: The automata corresponding to the two jobs $J^1 = (m_1, 4), (m_2, 5)$ and $J^2 = (m_1, 3)$.

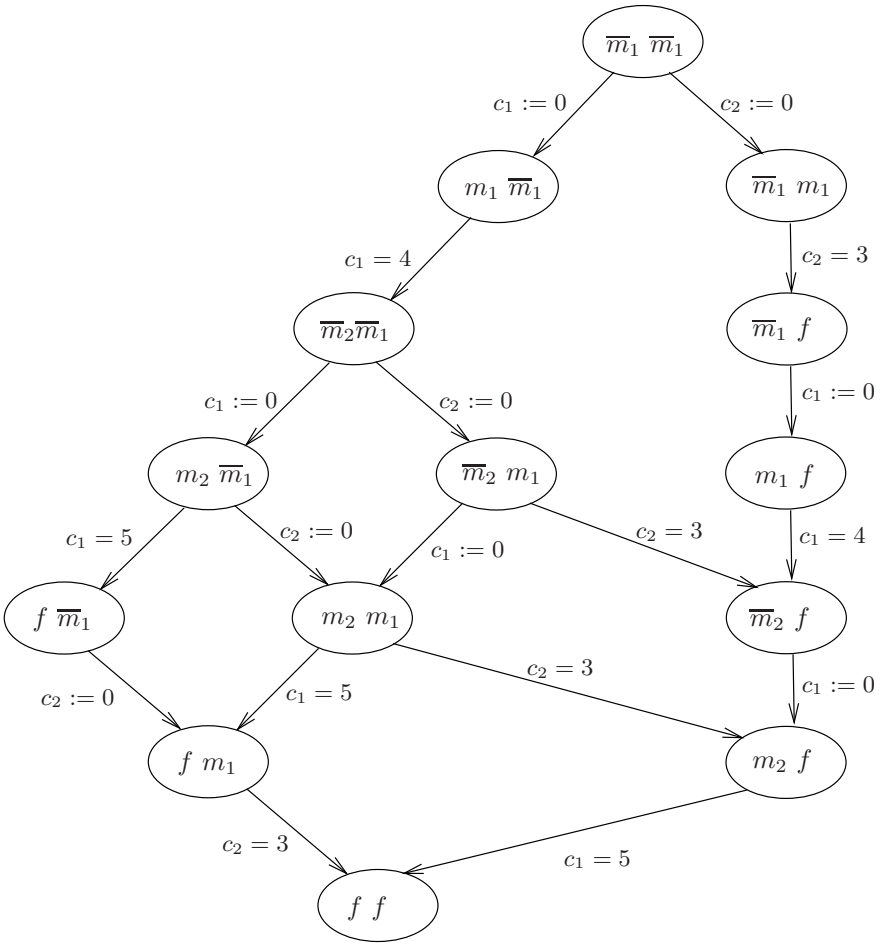


Figure 5.5: The global timed automaton for the two jobs.

5.3 Runs and Schedules

In this section we will show the tight correspondence between feasible schedules and runs of the automaton.

Definition 13 (Derived Schedule)
The derived schedule S_ξ *of a run* ξ *is a schedule where for every* i, j $\ s(i,j) = t$ *where* t *is the absolute time in* ξ *and* $\mathcal{A}^i$ *makes a start transition from* $\overline{\mu}^i(j)$ *to* $\mu^i(j)$

Claim 1 *Let* $\mathcal{A}$ *be the automaton generated for the job-shop specification* $\mathcal{J}$ *according to Definitions 1 and 2. Then:*

1. *For every complete run* ξ *of* $\mathcal{A}$*, its derived schedule* S_ξ *is feasible for* $\mathcal{J}$*.*
2. *For every feasible schedule* S *for* $\mathcal{J}$ *there is a complete run* ξ *of* $\mathcal{A}$ *such that* $S_\xi = S$*.*

Proof: The proof is by induction on the lengths of the run and schedule. A partial schedule S' is a schedule S restricted to an interval $[0, t]$. A partial run is a run not reaching f. The *section* of a schedule at time t is given by a tuple $(\overline{P}, P, \underline{P}, e)$ such that $\overline{P}$, P and $\underline{P}$ are, respectively,

the sets of waiting, active and finished steps and e is a function from P to $\mathbb{R}_+$ indicating the time elapsed since the beginning of each active state. Formally:

$$\begin{aligned}\overline{P} &= \{(i,j) : s(i,j) > t\}\\ \overline{\underline{P}} &= \{(i,j) : 0 \leq t - s(i,j) \leq d^i(j)\}\\ \underline{P} &= \{(i,j) : d^i(j) \leq t - s(i,j)\}\end{aligned}$$

and $e(i,j) = t - s(i,j)$. We define the following correspondence between configurations of the automaton and sections of a schedule. Let $((q_1,\ldots,q_n),(v_1,\ldots,v_n),t)$ be an extended configuration. Its associated section is defined by defining for each $i \in \mathcal{J}$

$$q_i = \overline{\mu}^i(j) \text{ iff } \begin{array}{l}\forall j' < j \ (i,j) \in \underline{P} \wedge\\ \forall j' \geq j \ (i,j) \in \overline{P}\end{array}$$

$$q_i = \mu^i(j) \text{ iff } \begin{array}{l}\forall j' < j \ (i,j) \in \underline{P} \wedge\\ \forall j' > j \ (i,j) \in \overline{P} \wedge\\ \qquad (i,j) \in P \wedge\\ v_i = t - s(i,j)\end{array}$$

The inductive hypothesis is that every partial run reaching $(q,\mathbf{v},t)$ corresponds to a partial schedule which is feasible until t and whose section at t matches $(q,\mathbf{v},t)$. This is true at the initial state and time $t = 0$ and can be easily shown to be preserved by discrete transitions and time passage. The proof of the other direction is similar, progressing over the ordered set of start and end time points. ▟

The schedules S_1 and S_2 appearing in Figures 5.6 and 5.7 correspond respectively to the complete runs ξ_1 and ξ_2 of the timed automaton of Figure 8.5.

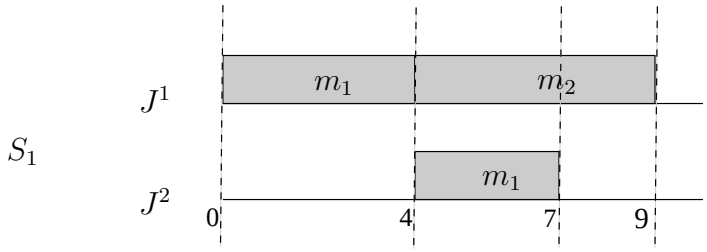


Figure 5.6: A schedule

$$\begin{aligned}\xi_1 : (\overline{m}_1,\overline{m}_1,\perp,\perp) &\xrightarrow{0} (m_1,\overline{m}_1,0,\perp) \xrightarrow{4} (m_1,\overline{m}_1,4,\perp) \xrightarrow{0} (\overline{m}_2,\overline{m}_1,\perp,\perp)\\ &\xrightarrow{0} (m_2,\overline{m}_1,0,\perp) \xrightarrow{0} (m_2,m_1,0,0) \xrightarrow{3} (m_2,m_1,3,3) \xrightarrow{0} (m_2,f,3,\perp)\\ &\xrightarrow{2} (m_2,f,5,\perp) \xrightarrow{0} (f,f,\perp,\perp)\end{aligned}$$

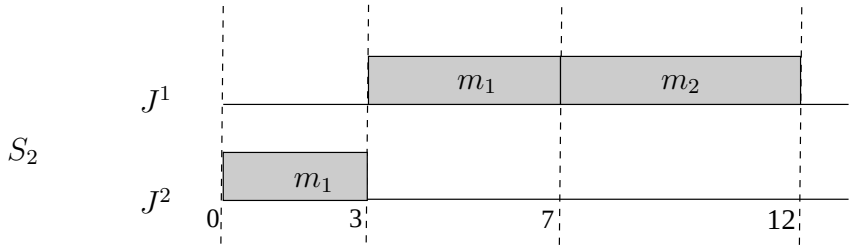


Figure 5.7: Another schedule

$$\xi_2 : (\overline{m}_1, \overline{m}_1, \bot, \bot) \xrightarrow{0} (\overline{m}_1, m_1, \bot, 0) \xrightarrow{3} (\overline{m}_1, m_1, \bot, 3) \xrightarrow{0} (\overline{m}_1, f, \bot, \bot)$$
$$\xrightarrow{0} (m_1, f, 0, \bot) \xrightarrow{4} (m_1, f, 4, \bot) \xrightarrow{0} (\overline{m}_2, f, \bot, \bot) \xrightarrow{0} (m_2, f, 0, \bot)$$
$$\xrightarrow{5} (m_2, f, 5, \bot) \xrightarrow{0} (f, f, \bot, \bot)$$

Corollary 2 (Job-Shop Scheduling and Timed Automata)
The optimal job-shop scheduling problem can be reduced to the problem of finding the shortest path in a timed automaton.

5.4 Shortest Path using Timed Automata Reachability

In this section we show how the symbolic forward reachability algorithm is used to find a shortest path and hence to solve the optimal job-shop scheduling problem.

Let $\mathcal{A}'$ be the extended timed automaton obtained from $\mathcal{A}$ by adding an absolute clock t. The two runs ξ_1' and ξ_2' are, respectively, the extended runs of ξ_1 and ξ_2.

$$\xi_1' : (\overline{m}_1, \overline{m}_1, \bot, \bot, 0) \xrightarrow{0} (m_1, \overline{m}_1, 0, \bot, 0) \xrightarrow{4} (m_1, \overline{m}_1, 4, \bot, 4) \xrightarrow{0} (\overline{m}_2, \overline{m}_1, \bot, \bot, 4)$$
$$\xrightarrow{0} (m_2, \overline{m}_1, 0, \bot, 4) \xrightarrow{0} (m_2, m_1, 0, 0, 4) \xrightarrow{3} (m_2, m_1, 3, 3, 7) \xrightarrow{0} (m_2, f, 3, \bot, 7)$$
$$\xrightarrow{2} (m_2, f, 5, \bot, 9) \xrightarrow{0} (f, f, \bot, \bot, 9)$$

$$\xi_2' : (\overline{m}_1, \overline{m}_1, \bot, \bot, 0) \xrightarrow{0} (\overline{m}_1, m_1, \bot, 0, 0) \xrightarrow{3} (\overline{m}_1, m_1, \bot, 3, 3) \xrightarrow{0} (\overline{m}_1, f, \bot, \bot, 3)$$
$$\xrightarrow{0} (m_1, f, 0, \bot, 3) \xrightarrow{4} (m_1, f, 4, \bot, 7) \xrightarrow{0} (\overline{m}_2, f, \bot, \bot, 7) \xrightarrow{0} (m_2, f, 0, \bot, 7)$$
$$\xrightarrow{5} (m_2, f, 5, \bot, 12) \xrightarrow{0} (f, f, \bot, \bot, 12)$$

The value of the additional clock in each reachable configuration represents the time to reach the configuration according to this run, and, in particular, its value in the final state represents the length of this run, 9 in ξ_1 and 12 in ξ_2.

Consequently to find the shortest path in a timed automaton we need to compare the value of t in all the reachable extended configurations of the form $(f, \mathbf{v}, t)$. This set of reachable configurations can be found using the standard forward reachability algorithm for acyclic timed automata Algorithm 4.

Remark: A common method used in reachability algorithms for reducing the number of explored symbolic states is the inclusion test. It is based on the fact that $Z \subseteq Z'$ implies

$Succ^{\delta}(q, Z) \subseteq Succ^{\delta}(q, Z')$ for every $\delta \in \Delta$. Hence, whenever a new symbolic state (q, Z) is generated, it is compared with any other (q, Z') in the waiting list: if $Z \subseteq Z'$ then (q, Z) is not inserted and if $Z' \subseteq Z$, (q, Z') is removed from the list. Allowing the job-shop automaton to stay indefinitely in any state makes the explored zones "upward-closed" with respect to absolute time and increases significantly the effectiveness of the inclusion test.

Figure 5.8 shows the simulation graph of the extended timed automaton of Figure 8.5. From every final symbolic state (f, Z) in the simulation graph we can extract $G(f, Z)$, the length of the minimal run among all the runs that share the same qualitative path:

$$G(f, Z) = \min\{t : (\mathbf{v}, t) \in Z\}$$

Thus the length of the optimal schedule is

$$t^* = \min\{G(f, Z) : (f, Z) \text{ is reachable in } \mathcal{A}'\}.$$

To construct the optimal schedule it is sufficient to find a run of length t^*. Hence, running a reachability algorithm on $\mathcal{A}'$ is guaranteed to find the minimal schedule.

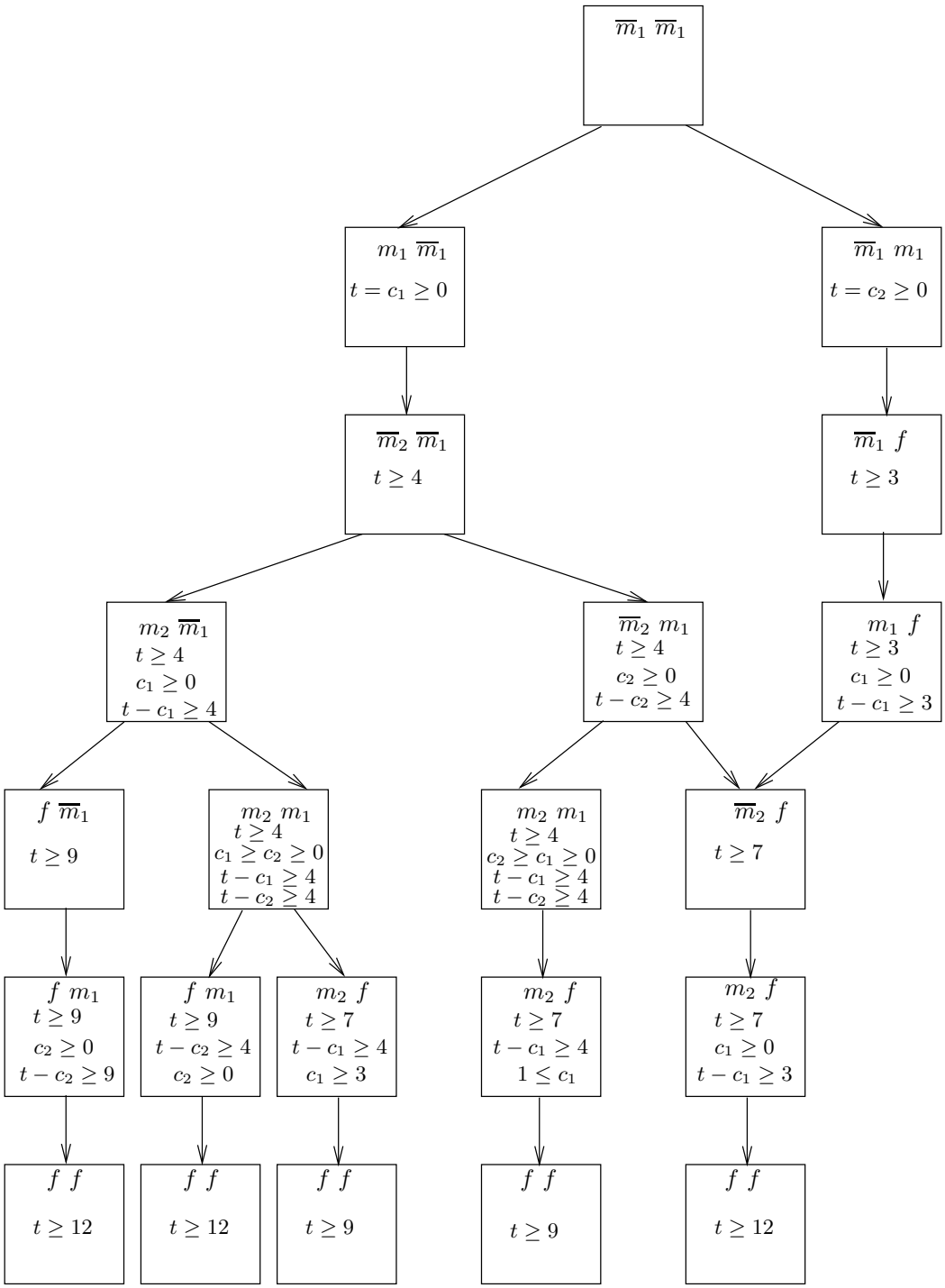


Figure 5.8: The simulation graph of the extended job-shop timed automaton of Figure 5.3

5.5 Laziness and Non-Laziness

In this section we show that only one point inside the zone of each symbolic state is sufficient for finding the optimal schedule.

5.5.1 Lazy Schedule

Definition 14 (Lazy Schedules)
Let S be a schedule, let i be a job and j a step with $\mu^i(j) = m$. We say that S exhibits laziness *at (i, j) if there exists in S an interval $[t, s(i, j)]$ where (i, j) is enabled. A schedule S is* non-lazy *if it exhibits no laziness.*

Laziness captures the phenomenon of useless waiting: a job whose step is enabled is waiting while no other job profits from its waiting. We will prove that every schedule can be transformed into a non-lazy one without increasing its length. Consider first the schedule S of Figure 5.9 exhibiting laziness at $(2, 1)$. By starting step $(2, 1)$ earlier we obtain the schedule S' with two new occurrences of laziness at $(2, 2)$ and $(3, 1)$. Those are removed yielding S'' with laziness at $(3, 2)$ and after removing it we obtain the non-lazy schedule $\hat{S}$.

Claim 3 (Non-Lazy Optimal Schedules) *Every lazy schedule S can be transformed into a non-lazy schedule $\hat{S}$ with $|\hat{S}| \leq |S|$. Hence every job-shop specification admits an optimal non-lazy schedule.*

Proof: Let S be a schedule, and let $L(S) \subseteq \mathcal{J} \times K$ be the set of steps that are not preceded by laziness in S, namely

$$L(S) = \{(i, j) : \forall (i', j') \preceq (i, j) \text{ there is no laziness in} (i', j')\}.$$

In Figure 5.9 these sets are at the left of the dashed lines. We pick a lazy step (i, j) s.t. $s(i, j) \leq s(i', j')$ for all $(i', j') \notin L(S)$ and shift its start time backward until we obtain a new feasible schedule S' which is not lazy at (i, j). The schedule S' verifies

$$L(S') \supseteq L(S) \cup (i, j)$$

and

$$|S'| \leq |S|.$$

Applying this procedure successively we increase $L(S)$ at each step and due to finiteness of the set of steps, the laziness removal procedure terminates. ∎

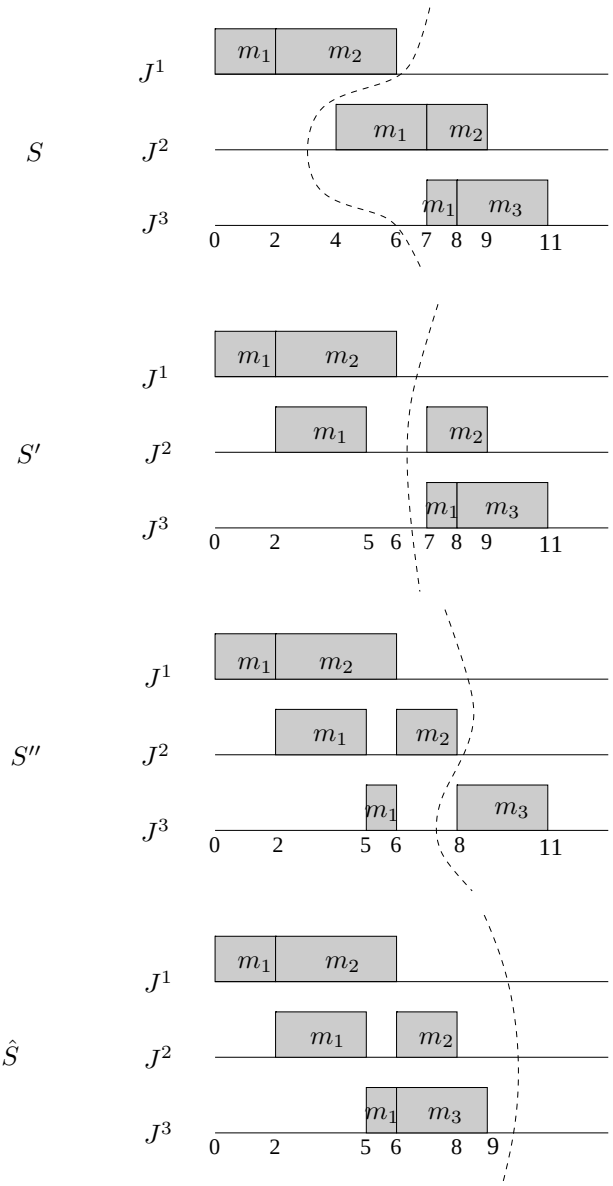


Figure 5.9: Removing laziness from a schedule. The dashed line indicates the frontier between $L(S)$ and the rest of the steps

5.5.2 Immediate and Lazy Runs

Having shown that an optimal schedule can be found among the non-lazy ones we can modify our search procedure to look only for runs of the automaton whose corresponding schedules are non-lazy. We first define the weaker notion of *immediate* runs such that each non-lazy schedule corresponds to an immediate run but not all immediate runs are non-lazy.

Definition 15 (Immediate Runs)
An immediate run *is a run where all the transitions are taken as soon as they are enabled. A* non-immediate run *contains a fragment*

$$(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t) \xrightarrow{0} (q', \mathbf{v}')$$

where the transition taken at $(q, \mathbf{v}+t)$ *is enabled already at* $(q, \mathbf{v}+t')$ *for some* $t' < t$.

Clearly a derived schedule of a non-immediate run exhibits laziness.

Corollary 4 *In order to find an optimal schedule it is sufficient to explore the (finite) set of immediate runs.*

Every qualitative path in a timed automaton may correspond to infinitely many runs. For example the family of runs $\{\xi(r) : 0 \le r \le 2\}$

$$\begin{aligned}\xi(r) : (\overline{m}_1, \overline{m}_1, \perp, \perp) &\xrightarrow{0} (m_1, \overline{m}_1, 0, \perp) \xrightarrow{4} (m_1, \overline{m}_1, 4, \perp) \xrightarrow{0} (\overline{m}_2, \overline{m}_1, \perp, \perp) \\ &\xrightarrow{0} (m_2, \overline{m}_1, 0, \perp) \xrightarrow{r} (m_2, \overline{m}_1, r, \perp) \xrightarrow{0} (m_2, m_1, r, 0) \xrightarrow{3} (m_2, m_1, r+3, 3) \\ &\xrightarrow{0} (m_2, f, r+3, \perp) \xrightarrow{2-r} (m_2, f, 5, \perp) \xrightarrow{0} (f, f, \perp, \perp)\end{aligned}$$

Each $\xi(r)$ corresponds to a schedule like $S(r)$ of Figure 5.10. Whenever $r > 0$ the run is not immediate and the schedule is lazy. Corollary 4 allows us to explore only $\xi(0)$.

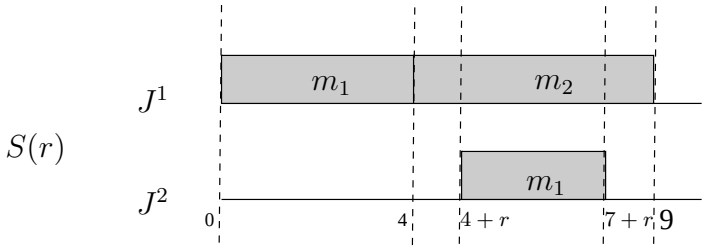


Figure 5.10:

Note that the derived schedule of an immediate run is not necessarily non-lazy. The feasible schedule of Figures 5.11 derived from run ξ_3 is a lazy schedule, while run ξ_3 is an immediate run.

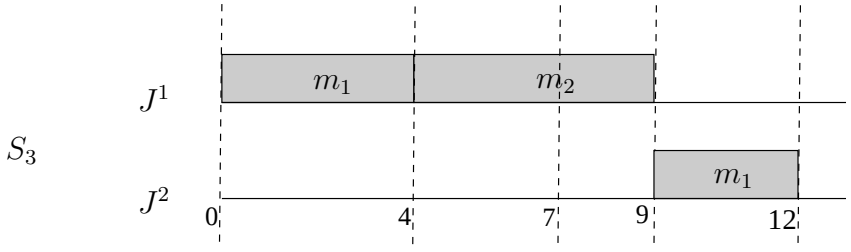


Figure 5.11:

$$\begin{aligned}\xi_3 : (\overline{m}_1, \overline{m}_1, \perp, \perp) &\xrightarrow{0} (m_1, \overline{m}_1, 0, \perp) \xrightarrow{4} (m_1, \overline{m}_1, 4, \perp) \xrightarrow{0} (\overline{m}_2, \overline{m}_1, \perp, \perp) \\ &\xrightarrow{0} (m_2, \overline{m}_1, 0, \perp) \xrightarrow{5} (m_2, \overline{m}_1, 5, \perp) \xrightarrow{0} (f, \overline{m}_1, \perp, 0) \xrightarrow{0} (f, m_1, \perp, 0) \\ &\xrightarrow{3} (f, m_1, 3, \perp) \xrightarrow{0} (f, f, 5, \perp)\end{aligned}$$

Definition 16 (Lazy Runs)
A lazy run *in a job-shop timed automaton $\mathcal{A}$ is a run containing a fragment*

$$(q, \mathbf{v}) \dots \xrightarrow{t} \dots (q', \mathbf{v}') \xrightarrow{start_i} (q'', \mathbf{v}'')$$

s.t. the $start_i$ transition is enabled in all states $(q, \mathbf{v}) \dots (q', \mathbf{v}')$.

The immediate run ξ_3 is lazy due to the fragment $(m_2, \overline{m}_1, 0, \perp) \xrightarrow{5} (m_2, \overline{m}_1, 5, \perp) \xrightarrow{0} (f, \overline{m}_1, \perp, 0) \xrightarrow{0} (f, m_1, \perp, 0)$. The start transition for J^2 is taken at $(f, \overline{m}_1)$ while it was continuously enabled since $(m_2, \overline{m}_1)$.

Claim 5 *Let S^* be the set of non-lazy schedules of a job shop specification $\mathcal{J}$ and let $\mathcal{A}$ be its automaton. Let S_I and S_L be the sets of derived schedules for all the immediate and non-lazy runs of $\mathcal{A}$, respectively, then*

$$S^* = S_L \subseteq S_I.$$

Corollary 6 (Job-Shop Scheduling and Timed Automata) *The optimal job-shop scheduling problem can be reduced to the problem of finding the shortest non-lazy path in a timed automaton.*

5.6 The Search Algorithm

Based on Corollary 11 we build a search algorithm that explores only the non-lazy runs. For more clarity we start by showing how to generate immediate runs, then reduce even more the search space to obtain the set of non-lazy runs.

Definition 17 (Domination point)
Let (q, Z) be a symbolic state in an extended timed automaton. We say that the reachable configuration $(q, \mathbf{v}^*, t^*)$ is domination point *of (q, Z) if*

$$t^* = G(q, Z) \quad \textit{(earliest arrival time)}$$

and for every i,

$$v_i^* = \max\{v_i : (v_1, \dots, v_i, \dots, v_n, t^*) \in Z\}.$$

This point is the one reachable via an immediate run. Restricting the search algorithm to these points and their successor makes the algorithm much more efficient. Instead of working with symbolic state of the form (q, Z) where Z is a zone represented by a DBM of size $O(n^2)$, and where computing successors is a non-trivial operation, we can work with a point representation of size $O(n)$ where passage of time is a simple vector addition.

We define the timed successor $Succ^t(q, \mathbf{v}, t)$ and the discrete successor $Succ^\delta(q, \mathbf{v}, t)$ of every configuration $(q, \mathbf{v}, t)$ as follows:

Let θ be the maximal amount of time that can elapse in a configuration $(q, \mathbf{v}, t)$ until an end transition becomes enabled, i.e.

$$\theta = \min\{(d^i - v_i) : c_i \text{ is active at } q\}.$$

The timed successor of a configuration is the result of letting time progress by θ and terminating all that can terminate by that time:

$$Succ^t(q_1, \dots, q_n, v_1, \dots, v_n, t) = \{(q_1', \dots, q_n', v_1', \dots, v_n', t + \theta)\}$$

such that for every i

$$(q'_i, v'_i) = \begin{cases} (q''_i, v''_i) & \text{if the transition } (q_i, v_i + \theta) \to (q''_i, v''_i) \text{ is enabled} \\ (q_i, v_i + \theta) & \text{otherwise.} \end{cases}$$

The discrete successors are all the successors by immediate start transition:

$$Succ^{\delta}(q, \mathbf{v}, t) = \{(q', \mathbf{v}', t) \text{ s.t. } (q, \mathbf{v}, t) \xrightarrow{\text{start}_i} (q', \mathbf{v}', t)\}$$

The set of successors of each $(q, \mathbf{v}, t)$ is:

$$Succ(q, \mathbf{v}, t) = Succ^{t}(q, \mathbf{v}, t) \cup Succ^{\delta}(q, \mathbf{v}, t)$$

Using the successor operator in a reachability algorithm for discrete graphs, we can compute all the immediate runs in a timed automaton. Figure 5.12 shows the 5 immediate runs of the automaton of Figure 8.5.

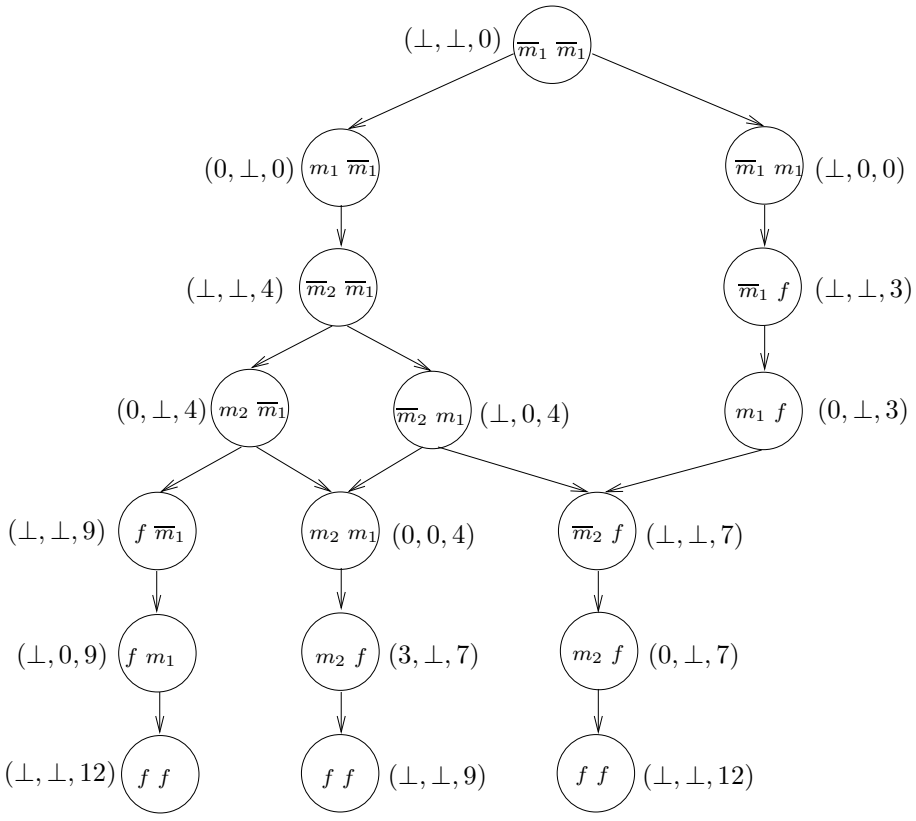


Figure 5.12: The immediate runs of the timed automaton of Figure 8.5

To restrict the search further to non-lazy runs, we must eliminates all useless waiting. This can be done as follows. If from a state $(q, \mathbf{v})$ we choose to take the time successor $(q, \mathbf{v} + \theta)$ while a start_i transition associated with a step (i, j) is enabled in $(q, \mathbf{v})$, we mark component i as "freezed" in $(q, \mathbf{v} + \theta)$ and this marking is propagated to its successors and is removed only after the $start_i$ transition was disabled and enabled again (some other job took and then released the machine in question). In every configuration we restrict the set of successors only to transitions of non-freezed components. Moreover, if the duration of step (i, j) is such that if started at $(q, \mathbf{v})$

it will terminate before another conflicting transition is enabled, it must be taken immediately and not freezed (this is the case when it is the only remaining step that needs the machine).

This successor computation is implemented using annotated configurations of the form $(q, \mathbf{v}, t, F)$ where F is a set of freezed components, initialized to the empty set. The successors are computed as follows:

$$Succ^t(q, \mathbf{v}, t, F) = (Succ^t(q, \mathbf{v}, t), F \cup F_1 - F_2)$$

where F_1 is the set of components i s.t. a start$_i$ transition is enabled in $(q, \mathbf{v}, t)$ and F_2 is the set of components i s.t. the transition releases the machine for which i is waiting, and

$$Succ^\delta(q, \mathbf{v}, t, F) = \{(q', \mathbf{v}', t, F) : (q, \mathbf{v}, t) \rightarrow (q', \mathbf{v}', t)\}$$

where $(q, \mathbf{v}, t) \rightarrow (q', \mathbf{v}', t)$ is a start$_i$ transition s.t. $i \notin F$.

If there is a transition enabled at $(q, \mathbf{v})$ which will not block any other job

$$Succ(q, \mathbf{v}, t, F) = Succ^\delta(q, \mathbf{v}, t, F)$$

otherwise

$$Succ(q, \mathbf{v}, t, F) = Succ^\delta(q, \mathbf{v}, t, F) \cup Succ^t(q, \mathbf{v}, t, F)$$

Applying these operators in a reachability algorithm, we obtain the set of non-lazy runs. Figure 5.14 shows the 4 non-lazy runs of the automaton of Figure 8.5.

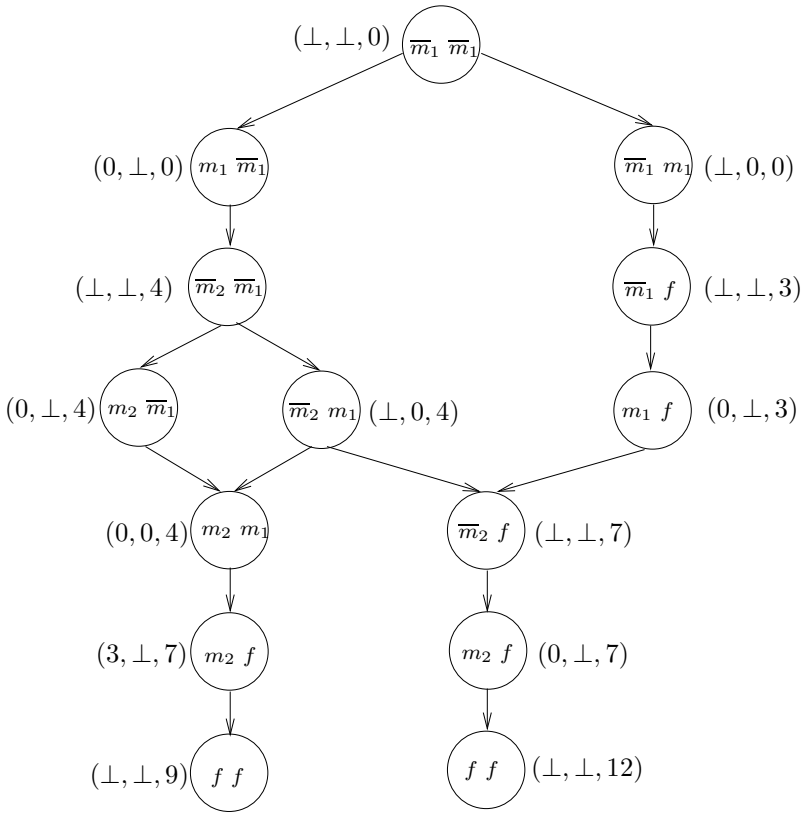


Figure 5.13: Non-lazy runs of the timed automaton of Figure 8.5

5.7 Reducing the Search Space

Although using points instead of zones reduces significantly the computational cost, the inherent combinatorial explosion remains. In this section we describe further methods to reduce the search spaces, some of which preserve the optimal solutions and some provide sub-optimal ones.

5.7.1 Domination Test

The configurations $(m_2, f, 3, \perp, 7)$ and $(m_2, f, 0, \perp, 7)$ in Figure 5.14 share the same state (m_1, f) but have different clock values. Both are reached at the same time $t = 7$ but in $(m_2, f, 0, \perp, 7)$ the value of c_1 is smaller, and hence all the run continuing from it cannot reach f before those continuing from $(m_2, f, 3, \perp, 7)$. Hence the successors of $(m_2, f, 0, \perp, 7)$ can be discarded without missing the optimum.

Definition 18 (Domination)
Let $(q, \mathbf{v}, t)$ *and* $(q, \mathbf{v}', t')$ *be two reachable configurations. We say that* $(q, \mathbf{v}, t)$ dominates $(q, \mathbf{v}', t')$ *if* $t' \leq t$ *and* $\mathbf{v} \geq \mathbf{v}'$.

Clearly if $(q, \mathbf{v}, t)$ dominates $(q, \mathbf{v}', t')$ then for every complete run going through $(q, \mathbf{v}', t')$ there is a run which traverses $(q, \mathbf{v}, t)$ and which is not longer.

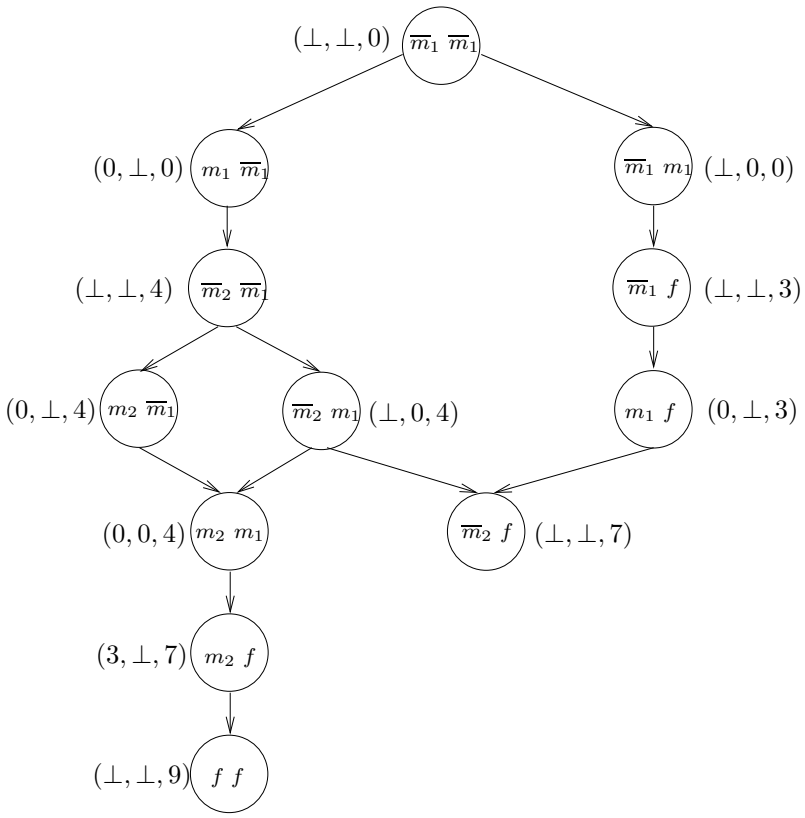


Figure 5.14:

5.7.2 Best-first Search

In order to apply best-first search and explore the "most promising" directions first, we need an evaluation function over configurations. For every configuration (q, v) of a job automaton $g(q, v)$ is a lower-bound on the time remaining until f is reached from the configuration (q, v):

$$\begin{aligned} &g(f, v) = 0 \\ &g(\overline{\mu}(j), v) = \textstyle\sum_{l=j}^{k} d(l) \\ &g(\mu(j), v) = g(\overline{\mu}(j), v) - \min\{v, d(j)\} \end{aligned}$$

The evaluation of global configurations is defined as:

$$E((q_1, \ldots, q_n), (v_1, \ldots, v_n, t)) = t + \max\{g^i(q_i, v_i)\}_{i=1}^{n}$$

Note that $\max\{g^i\}$ gives the most optimistic estimation of the *remaining* time, assuming that no job will have to wait. The best-first search algorithm is guaranteed to produce the optimal path because it stops the exploration only when it is clear that the unexplored states cannot lead to schedules better than those found so far.

Algorithm 6 (Best-first Forward Reachability)
Waiting:=$\{Succ(s, \mathbf{v}, t)\}$*;*
Best:=∞
$(q, \mathbf{v}, t)$*:= first in Waiting;*
while *Best* $> E(q, \mathbf{v}, t)$
do
  For every $(q', \mathbf{v}, t) \in Succ(q, \mathbf{v}, t)$*;*
    if $q' = f$ **then**
      Best:=$\min\{Best, E(q', \mathbf{v}, t)\}$
    else
      Insert $(q', \mathbf{v}', t)$ *into Waiting;*
  Remove $(q, \mathbf{v}, t)$ *from Waiting*
  $(q, \mathbf{v}, t)$*:= first in Waiting;*
end

5.7.3 Sub-Optimal Solutions

The best-first algorithm improves performance but the combinatorial explosion remains. To avoid it we use an algorithm which is a mixture of breadth-first and best-first search with a fixed number w of explored nodes at any level of the automaton. For every level we take the w best (according to E) states, generate their successors but explore only the best w among them, and so on. The number w is the main parameter of this technique, and although the number of explored states grows monotonically with w, the quality of the solution does not — sometimes the solution found with a smaller w is better than the one found with a larger w.

5.8 Experimental Results

We have implemented a prototype that models the job shop problem in a timed automaton, and generates all the non-lazy runs. Applying the domination test and using a best first search we could solve problems with 6 jobs and 6 machines in few seconds. Beyond that we had to employ the sub-optimal heuristic.

We tested the heuristic algorithm on 10 problems among the most notorious job-shop scheduling problems. Note that these are pathological problems with a large variability in step durations, constructed to demonstrate the hardness of job-shop scheduling. For each of these problems we have applied our algorithm for different choices of w. In Table 6.4 we compare our best results on these problems with the best results reported in Table 15 of the recent survey [JM99], where the results of the 18 best-known methods were compared. As one can see our results are typically $5 - 10\%$ from the optimum.

problem			heuristic			Opt
name	#j	#m	time	length	deviation	length
FT10	10	10	3	969	4.09 %	930
LA02	10	5	1	655	0.00 %	655
LA19	10	10	15	869	3.21 %	842
LA21	10	15	98	1091	4.03 %	1046
LA24	10	15	103	973	3.95 %	936
LA25	10	15	148	1030	5.42 %	977
LA27	10	20	300	1319	6.80 %	1235
LA29	10	20	149	1259	9.29 %	1152
LA36	15	15	188	1346	6.15 %	1268
LA37	15	15	214	1478	5.80 %	1397

Table 5.1: The results for 10 hard problems using the bounded width heuristic. The first three columns give the problem name, no. of jobs and no. of machines (and steps). Our results (time in seconds, the length of the best schedule found and its deviation from the optimum) appear next,

To appreciate the contribution of the fact that we use points instead of zones due to non-laziness, we can look at he performance of zone-based versions of our algorithms et Table 5.2, where the largest problem that can be solved exactly is of size 6 jobs and 4 machines.

Problem size			Inclusion		Domination		Best-first	
#j	#ds	#tree	#s	time	#s	time	#s	time
2	77	632	212	1	100	1	38	1
3	629	67298	5469	2	1143	1	384	1
4	4929	279146	159994	126	11383	2	1561	1
5	37225	m.o.	m.o.	m.o.	116975	88	2810	1
6	272125	m.o.	m.o.	m.o.	1105981	4791	32423	6

Table 5.2: The results for n jobs with 4 tasks. Columns #j, #ds and #tree show, respectively, the number of jobs, the number of discrete states in the automaton and the number of different reachable symbolic states (which is close to the number of nodes in the unfolding of the automaton into a tree). The rest of the table shows the performance, in terms of the number of explored symbolic states and time (in seconds), of algorithms employing, progressively, the inclusion test, the domination test, and the best-first search (m.o. indicates memory overflow).

Chapter 6

Preemptive Job Shop Scheduling

In this chapter we extend the results on deterministic Job-shop scheduling problem to preemptible jobs, i.e. jobs that can use a machine for some time, stop for a while and then resume from where they stopped. Such situations are common, for example, when the machines are computers.

As an example let $M = \{m_1, m_2, m_3\}$ be a set of resources and $\mathcal{J} = \{J^1, J^2\}$ a job specification over a M where

$$J^1 = (m_1, 3), (m_2, 2), (m_3, 4) \text{ and } J^2 = (m_2, 5).$$

Two feasible schedules S_1 and S_2 appear in Figure 6.1. In the schedule S_1 the job J^2 is preempted at time $t = 3$ on the machine m_2 to give the machine to the job J^1. After J^1 has terminated job J^2 restarts on m_2. The length of S_1 is 9 and it is the optimal schedule.

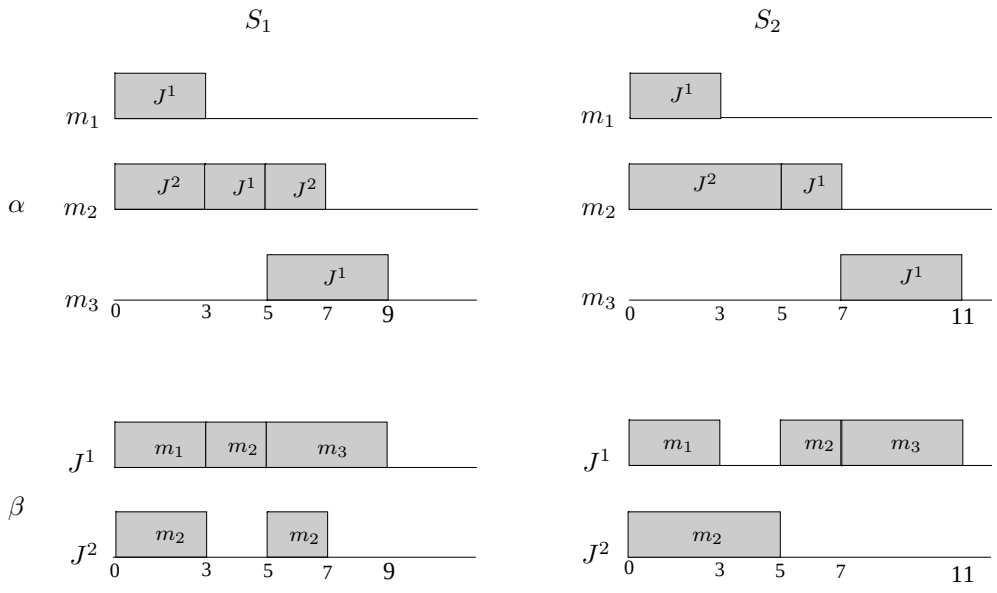


Figure 6.1: Two schedule S_1 and S_2 visualized as the machine allocation function α and the task progress function β.

The definition of a job-shop specification (Definition 1) and of feasible schedule (Definition 2) remains the same except the relaxation of the non preemption constraint in Definition 2. this

means that for every step (i, j) the set $\{t, (i, j, t) \in S\}$ is a union of intervals such that the sum of their lengths is equal to the step duration.

6.1 Modeling with Stopwatch Automata

The timed automaton model proposed in the Chapter 1, is not valid any more because it can not express preemption of a step. A job automaton can leave an execution state only if the step has terminate.

To model preemption we need an additional state "preempt" such that the automaton can move back and forth between "execute" and "preempt" as in Figure 6.2.

Since only the time spent in "execute" counts, we cannot use the value of c_1 as is in the transition guard to termination. One solution is to add an additional clock c_2, measuring the preemption time and updating c_1 to be $c_1 - c_2$ each time execution resumes. This operation is beyond the reset operation allowed in timed automata.

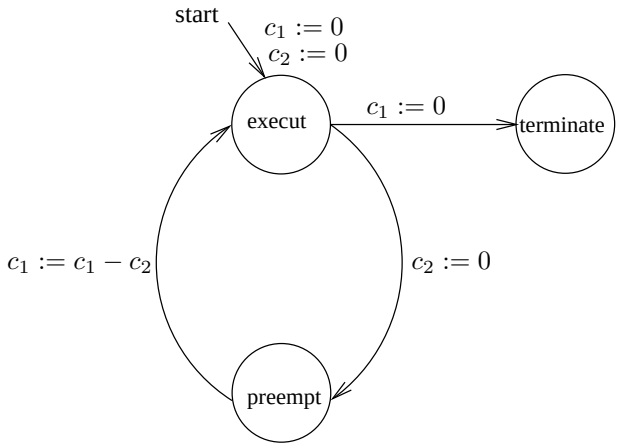


Figure 6.2:

An alternative and more natural solution is to extend the model of timed automata to have clocks which can be freezed at certain states, in other words clocks with derivative zero. Such automata were called Integration Graphs in [KPSY99] where they were studied as models for the Duration Calculus [CHR91]. The results in [KPSY99] included the undecidability of the reachability problem for these automata, and a decision procedure for some special sub-classes based on reducing the problem into linear constraint satisfaction. Similar automata were also investigated in [MV94] and in [CL00] where an implementation of an approximate verification algorithm was described.

Definition 19 (Stopwatch Automata)
A stopwatch automaton is a tuple $\mathcal{A} = (Q, C, s, f, \mathbf{u}, \Delta)$ where Q is a finite set of states, C is a finite set of n clocks, $\mathbf{u} : Q \to \{0, 1\}^n$ assigns a constant slope to every state and Δ is a transition relation consisting of elements of the form (q, ϕ, ρ, q') where q and q' are states, $\rho \subseteq C$ and ϕ (the transition guard) is a boolean combination of formulae of the form $(c \in I)$ for some clock c and some integer-bounded interval I. States s and f are the initial and final states, respectively.

6.1.1 Modeling Jobs

We construct for every job $J = (k, \mu, d)$ a stopwatch automaton with one clock such that for every step j with $\mu(j) = m$ there are three states: a waiting state $\overline{m}$, an active state m and a state $\tilde{m}$ indicating that the job is preempted after having started. Upon entering m the clock is reset to zero, and measures the time spent in m. Preemption and resumption are modeled by transitions to and from state $\tilde{m}$ in which the clock does not progress. When the clock value reaches $d(j)$ the automaton can leave m to the next waiting state. Let $\overline{M} = \{\overline{m} : m \in M\}$, $\tilde{M} = \{\tilde{m} : m \in M\}$ and let $\overline{\mu} : K \to \overline{M}$ and $\tilde{\mu} : K \to \tilde{M}$ be auxiliary functions such that $\overline{\mu}(j) = \overline{m}$ and $\tilde{\mu}(j) = \tilde{m}$ whenever $\mu(j) = m$.

Definition 20 (Stopwatch Automaton for a Job)
Let $J = (k, \mu, d)$ be a job. Its associated automaton is $\mathcal{A} = (Q, \{c\}, u, \Delta, s, f)$ with $Q = P \cup \overline{P} \cup \tilde{P} \cup \{f\}$ where $P = \{\mu(1), \ldots \mu(k)\}$,$\overline{P} = \{\overline{\mu}(1), \ldots, \overline{\mu}(n)\}$ and $\tilde{P} = \{\tilde{\mu}(1), \ldots, \tilde{\mu}(n)\}$. The slope is defined as $u_q = 1$ when $q \in P$ and $u_q = 0$ otherwise. The transition relation Δ consists of the following types of tuples

type	q	ϕ	ρ	q'	
1) begin	$\overline{\mu}(j)$	*true*	$\{c\}$	$\mu(j)$	$j = 1..k$
2) pause	$\mu(j)$	*true*	$\emptyset$	$\tilde{\mu}(j)$	$j = 1..k$
3) resume	$\tilde{\mu}(j)$	*true*	$\emptyset$	$\mu(j)$	$j = 1..k$
4) end	$\mu(j)$	$c = d(j)$	$\emptyset$	$\overline{\mu}(j+1)$	$j = 1..k-1$
end	$\mu(k)$	$c = d(k)$	$\emptyset$	f	

The initial state is $\overline{\mu}(1)$.

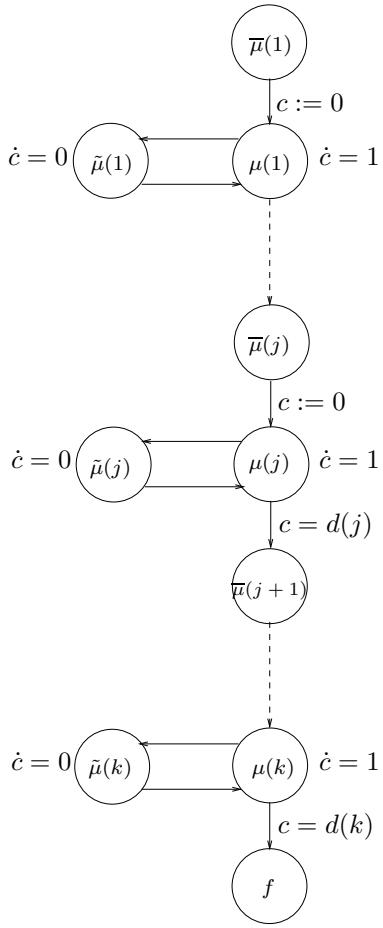


Figure 6.3: A generic stopwatch automaton for a job

The stopwatch automata corresponding to the two jobs of the Example are depicted in Figure 6.4.

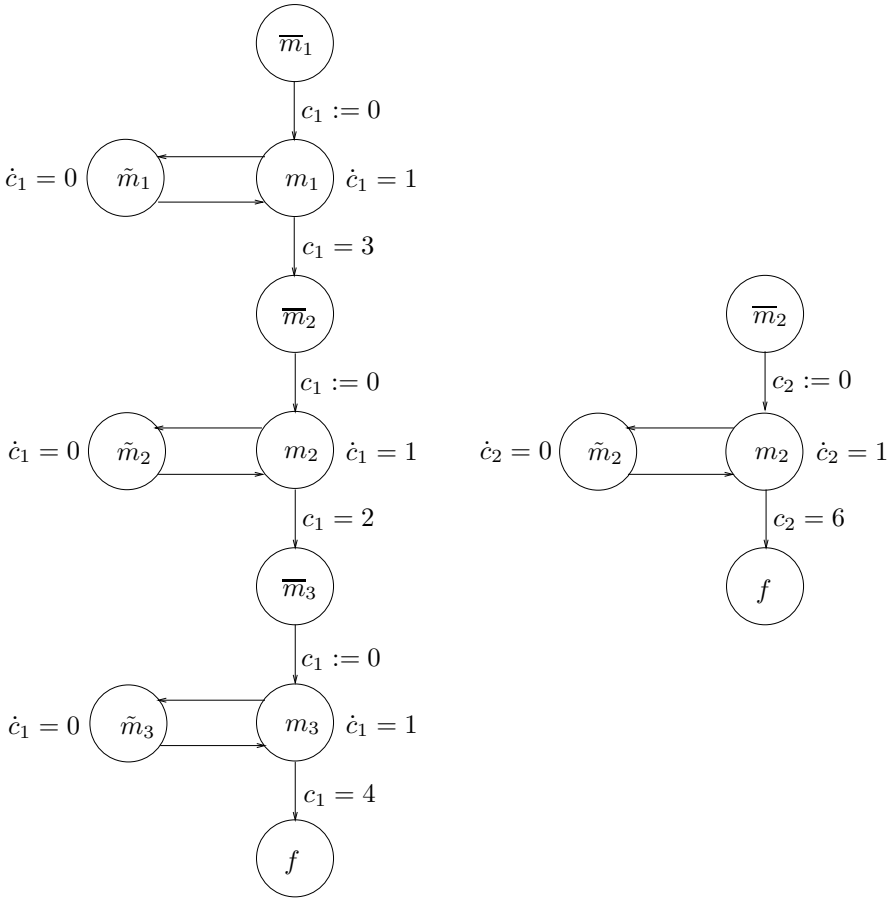


Figure 6.4: The automata corresponding to the jobs $J^1 = (m_1, 3), (m_2, 2), (m_3, 4)$ and $J^2 = (m_2, 5)$.

6.1.2 The Global Model

To obtain the stopwatch automaton modeling the preemptive job shop problem we need to compose the automata of the jobs, using the mutual exclusion composition described in the Chapter 1.

Part of the automaton obtained by composing the two automata of Figure 6.4 appears in Figure 6.5. We have omitted the preemption/resumption transitions for m_1 and m_3 as well as some other non-interesting paths. Unlike the non-preemptive job-shop automaton, this automaton is cyclic, due to the possibility to preempt and resume a step at any moment.

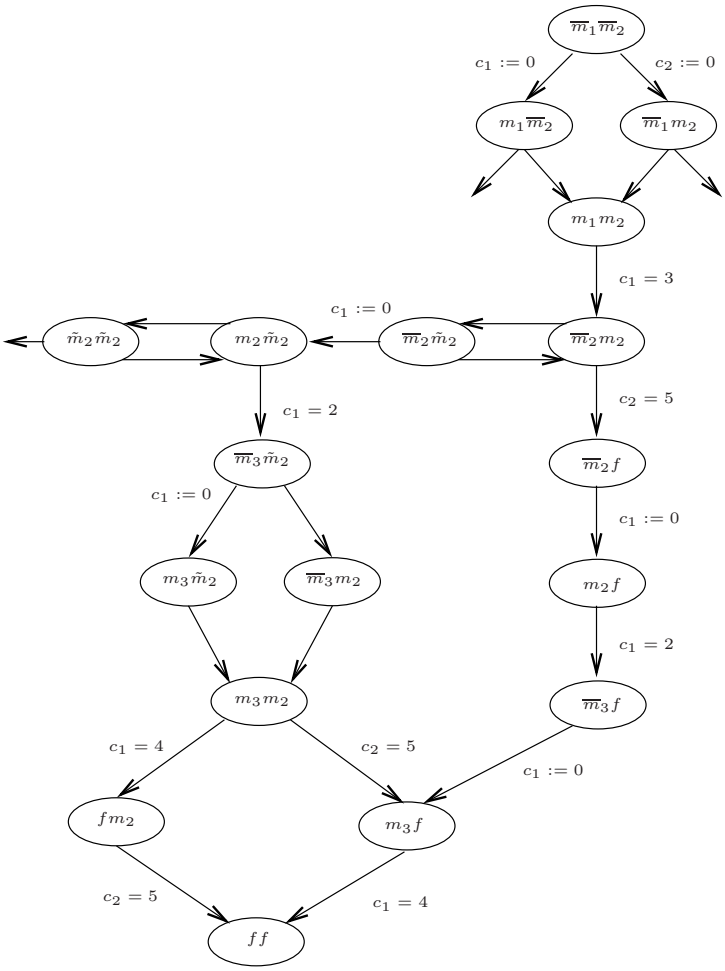


Figure 6.5: Part of the global stopwatch automaton for the two jobs.

6.1.3 Runs and Schedules

The correspondence between runs and feasible schedules is similar to the non-preemptive problem.

Claim 7 *Let $\mathcal{A}$ be the stopwatch automaton of a preemptive job shop specification $\mathcal{J}$. Every complete run of $\mathcal{A}$ corresponds to a feasible schedule with a length equal to the metric length of the run.*

The two schedules of Figure 6.1 correspond to the following two runs:

ξ_1 :
$(\overline{m}_1, \overline{m}_2, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_2, 0, \perp) \xrightarrow{0} (m_1, m_2, 0, 0) \xrightarrow{3} (m_1, m_2, 3, 3) \xrightarrow{0}$
$(\overline{m}_2, m_2, \perp, 3) \xrightarrow{0} (\overline{m}_2, \tilde{m}_2, \perp, 3) \xrightarrow{0} (m_2, \tilde{m}_2, 0, 3) \xrightarrow{2} (m_2, \tilde{m}_2, 2, 3) \xrightarrow{0}$
$(\overline{m}_3, \tilde{m}_2, \perp, 3) \xrightarrow{0} (\overline{m}_3, m_2, \perp, 3) \xrightarrow{0} (m_3, m_2, 0, 3) \xrightarrow{2} (m_3, m_2, 2, 5) \xrightarrow{0}$
$(m_3, f, 2, \perp) \xrightarrow{2} (m_3, f, 4, \perp) \xrightarrow{0} (f, f, \perp, \perp)$

ξ_2 :
$(\overline{m}_1, \overline{m}_2, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_2, 0, \perp) \xrightarrow{0} (m_1, m_2, 0, 0) \xrightarrow{3} (m_1, m_2, 3, 3) \xrightarrow{0}$
$(\overline{m}_2, m_2, \perp, 3) \xrightarrow{2} (\overline{m}_2, m_2, \perp, 5) \xrightarrow{0} (\overline{m}_2, f, \perp, \perp) \xrightarrow{0} (m_2, f, 0, \perp) \xrightarrow{2}$
$(m_2, f, 2, \perp) \xrightarrow{0} (\overline{m}_3, f, \perp, \perp) \xrightarrow{0} (m_3, f, 0, \perp) \xrightarrow{4} (m_3, f, 4, \perp) \xrightarrow{0}$
$(f, f, \perp, \perp)$

Corollary 8
The optimal preemptive job-shop scheduling problem can be reduced to the problem of finding the shortest path in a stopwatch automaton.

While trying to find the shortest path in this automaton we encounter two problems:

1. General reachability problems for stopwatch automata are known to be undecidable [C92, KPSY99].

2. The global stopwatch automaton is cyclic and thus have an *infinite* number of qualitative runs.

However, we will show, using a well-known result concerning optimal preemptive schedules, that these problems can be overcome.

6.2 Efficient Schedules

Definition 21 (Conflicts and Priorities)
Let S be a feasible schedule. let T^i_j be the set of time instants where job $i \in \mathcal{J}$ is executing its j^{th} step and $\mathcal{E}^i_j = [s(i, j-1) + d^i(j-1), s(i, j) + d^i(j)]$ i.e. the time interval between the enabling of the step and its termination. We say that job i is in conflict with job i' on machine m in S (denoted by $i \not\parallel_m i'$) when there are two respective steps j and j' such that $\mu^i(j) = \mu^{i'}(j') = m$ and $\mathcal{E}^i_j \cap \mathcal{E}^{i'}_{j'} \neq \emptyset$. We say that i has priority in S on m over a conflicting job i' (denoted by $i \prec_m i'$) if it finishes using m before i' does, i.e. $s(i, j) + d^i(j) < s(i', j') + d^{i'}(j')$.

Definition 22 (Efficient Schedules)
A schedule S is efficient if for every job i and a step j such that $\mu^i(j) = m$, job i uses m during all the time interval $\mathcal{E}^i_j$ except for times when another job i' such that $i' \prec_m i$ uses it.

In other words, efficiency means that every step (i, j) uses the machine during all $\mathcal{E}^i_j$ except for times when the machine is used by a step which terminates earlier than (i, j).

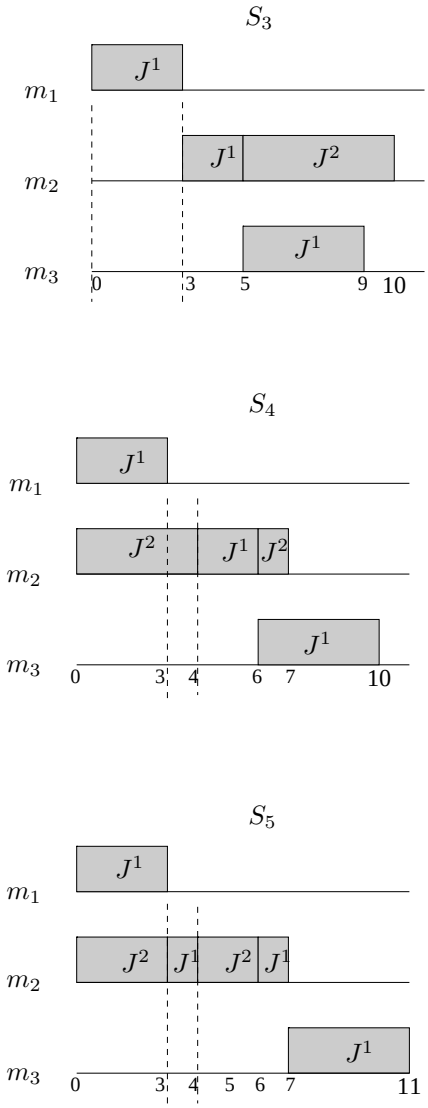


Figure 6.6: Three inefficient schedule S_3, S_4 and S_5. The interval of inefficiency are indicated by the dashed lines

The schedules of Figure 6.6 are three feasible schedules of the example. We will see why these three schedules are all inefficient schedules. According to Definition 21 we can deduce for each problem the priority relation between the jobs J^1 and J^2 on machine m_2 and the time intervals $\mathcal{E}_2^1$ and $\mathcal{E}_1^2$.

- $S_3 : J^1 \prec_{m_2} J^2$, $\mathcal{E}_2^1 = [3, 5]$, $\mathcal{E}_1^2 = (0, 10]$
- $S_4 : J^1 \prec_{m_2} J^2$, $\mathcal{E}_2^1 = [3, 6]$, $\mathcal{E}_1^2 = (0, 7]$
- $S_5 : J^2 \prec_{m_2} J^1$, $\mathcal{E}_2^1 = [3, 7]$, $\mathcal{E}_1^2 = (0, 6]$

In schedule S_3 job J^2 can use machine m_2 during the time interval $[0, 10]$ while it starts at $t = 5$ and no other job uses the machine in $[0, 3]$.

In schedule S_4 job J^1 can use machine m_2 during the time interval $[3, 6]$ while in $[3, 4]$, job J^2 occupies the machine while $J^1 \prec_{m_2} J^2$.

In schedule S_5 job J^2 can use machine m_2 during the time interval $[0, 6]$ while in $[3, 4]$, job J^1 occupies the machine while $J^2 \prec_{m_2} J^1$.

We will show that any problem admits an optimal schedule that corresponds to fixed priority relation among jobs on machine such that:

- A step of a job executes as soon as it is enabled except for times when it is in conflict with a higher priority job.
- Preemption occurs only when a step which has higher priority than an executing step becomes enabled. Hence the number of preemptions is finite.

Theorem 9 (Efficiency is Good) *Every preemptive job-shop specification admits an efficient optimal schedule.*

Proof: The proof is by showing that every inefficient schedule S can be transformed into an efficient schedule S' with $|S'| \leq |S|$. Let I be the first interval when inefficiency occurs for job J^i and machine m. We modify the schedule by shifting some of the later use of m by J^i into I. If m was occupied during I by another job $J^{i'}$ such that $J^i \prec_m J^{i'}$, we give it the time slot liberated by J^i. The termination of the step by $J^{i'}$ is not delayed by this modification because it happens anyway after J^i terminates its step. ∎

As an illustration consider the schedules appearing in Figure 6.7 with $J^1 \prec_m J^2 \prec_m J^3$ and where J^2 is enabled in the interval $[t_1, t_2]$. The first inefficiency in S_1 is eliminated in S_2 by letting J^2 use the free time slot before the arrival of J^1. The second inefficiency occurs when J^3 uses the machine while J^2 is waiting, and it is removed in S_3. The last inefficiency where J^3 is waiting while m is idle is removed in S_4.

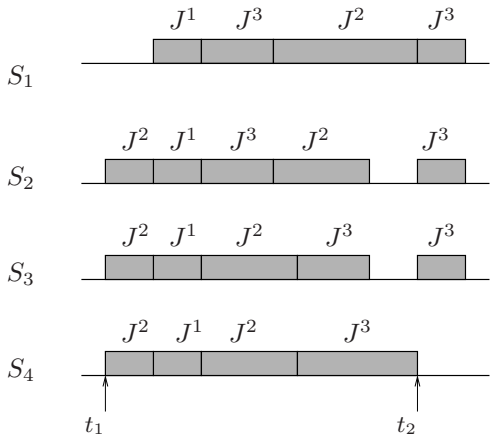


Figure 6.7: Removal of inefficiency, $J^1 \prec J^2 \prec J^3$.

This result reduces the set of candidates for optimality from the non-countable set of feasible schedules to the finite set of efficient schedules, each of which corresponds to a fixed priority relation. There are potentially $kn!$ priority relations but only a fraction of those needs to be considered because when i and i' are never in conflict concerning m, the priority $i \prec_m i'$ has no influence on the schedule.

6.3 Searching for Efficient Runs

In order to find shortest paths in stopwatch automata we will take advantage of Theorem 9 to restrict the search to runs whose corresponding schedules are efficient.

Definition 23 (Efficient Runs) *A run of a stopwatch automaton is efficient if all discrete transitions are taken as soon as they are enabled, and all conflicts are resolved according to a fixed priority relation.*

An efficient run allow us :

1. To restrict the search only to immediate runs.
2. To restrict the number of qualitative paths to be finite by avoiding loops and other useless preemptions and resumption.

To be more precise, let J^1 and J^2 be two jobs which are in conflict concerning machine m and let J^1 be the one with the highest priority on m. Table 6.3 depicts all the potential conflict situations and how they are resolved.

	state	action	new state	remark
1	$(\overline{m}, \overline{m})$	start 1	$(m, \overline{m})$	
2	$(\overline{m}, \tilde{m})$	start 1	$(m, \tilde{m})$	
3	$(\overline{m}, m)$	preempt 2	$(\overline{m}, \tilde{m})$	
4	$(\tilde{m}, \overline{m})$	resume 1	$(m, \overline{m})$	
5	$(\tilde{m}, \tilde{m})$	resume 1	$(m, \tilde{m})$	
6	$(\tilde{m}, m)$			(impossible)
7	$(m, \overline{m})$	(continue)	$(m, \overline{m})$	
8	$(m, \tilde{m})$	(continue)	$(m, \tilde{m})$	
9	(m, m)			(impossible)

Table 6.1: Resolving conflicts when $J^1 \preceq_m J^2$.

In situations 1, 2, 4, and 5 J^1 is waiting for the machine which is not occupied and so it takes it. Such situations could have been reached, for example, by a third job of higher priority releasing m or by J^1 finishing its prior step and entering $\overline{m}$. Situation 3 is similar but with J^2 occupying m and hence has to be preempted to reach situation 2. Situation 6, where J^1 is preempted and J^1 is executing, contradicts the priority and is not reachable. In situations 7 and 8, J^1 is executing and no preemption action is taken. Finally situation 9 violates mutual exclusion.

Claim 10 *Let $\mathcal{A}$ be the stopwatch automaton of a preemptive job shop specification $\mathcal{J}$. Every complete efficient run of $\mathcal{A}$ corresponds to a feasible efficient schedule with a length equal to the metric length of the run.*

Corollary 11 (Preemptive Scheduling and Stopwatch Automata)
The optimal preemptive job-shop scheduling problem can be reduced to the problem of finding the shortest efficient run in a stopwatch automaton.

The restriction to efficient runs makes the shortest path problem decidable: we can enumerate all the priority relation, and for each of them check the length of the induced efficient run.

As in the non-preemptive case, the search algorithm that we employ on the unfolding of the automaton generates priorities *on the fly* whenever two jobs come into conflict. In the example of Figure 6.5 the first conflict is encountered in state $(\overline{m}_2, m_2)$ and from there we may choose between two options, either to continue with time passage or preempt J^2. In the first case we fix the priority $J^2 \prec_{m_2} J^1$ and let J^2 finish without considering preemption anymore while in the second case the priority is $J^1 \prec_{m_2} J^2$, we move to $(\overline{m}_2, \tilde{m}_2)$ and the transition back to $(\overline{m}_2, m_2)$ becomes forbidden. From there we can only continue to $(m_2, \tilde{m}_2)$ and let the time pass until J^1 releases m_2.

To formalize this we define a *valid successors* relation over tuples of the form $(q, \mathbf{v}, t, \Pi)$ where $(q, \mathbf{v}, t)$ is a global configuration of the extended automaton and Π is a (partial) priority relation.

When there are no start transitions enabled in $(q, \mathbf{v}, t)$ we have

$$Succ(q, \mathbf{v}, t, \Pi) = \{(Succ^t(q, \mathbf{v}, t), \Pi)\}$$

where $Succ^t(q, \mathbf{v}, t)$ is the timed successor as defined for the non-preemptive case.

When there are start transitions enabled in $(q, \mathbf{v}, t)$ we have

$$Succ(q, \mathbf{x}, \Pi, \theta) = L_1 \cup L_2 \cup L_3$$

where

$$L_1 = \{(q', \mathbf{x}', \Pi, \theta) : (q, \mathbf{x}) \xrightarrow{\delta} (q', \mathbf{x}')\}$$

for every immediate transition δ such that δ is non-conflicting or belongs to the job whose priority on the respective machine is higher than those of all competing jobs. In addition, if there is a conflict on m involving a new job i whose priority compared to job i^*, having the highest priority so far, has not yet been determined, we have

$$L_2 = \{(q, \mathbf{x}, \Pi \cup \{i^* \prec i\}, \theta)\}$$

and

$$L_3 = \{(q, \mathbf{x}, \Pi \cup \bigcup_{\{i' : i' \not\parallel_m i\}} \{i \prec i'\}, \theta)\}.$$

The successor in L_2 represent the choice to prefer i^* over i (the priority of i relative to other waiting or preempted jobs will be determined only after i^* terminates), while L_3 represents the choice of preferring i over all other jobs.

The search tree generated by our algorithm for the example appears in Figure 6.8.

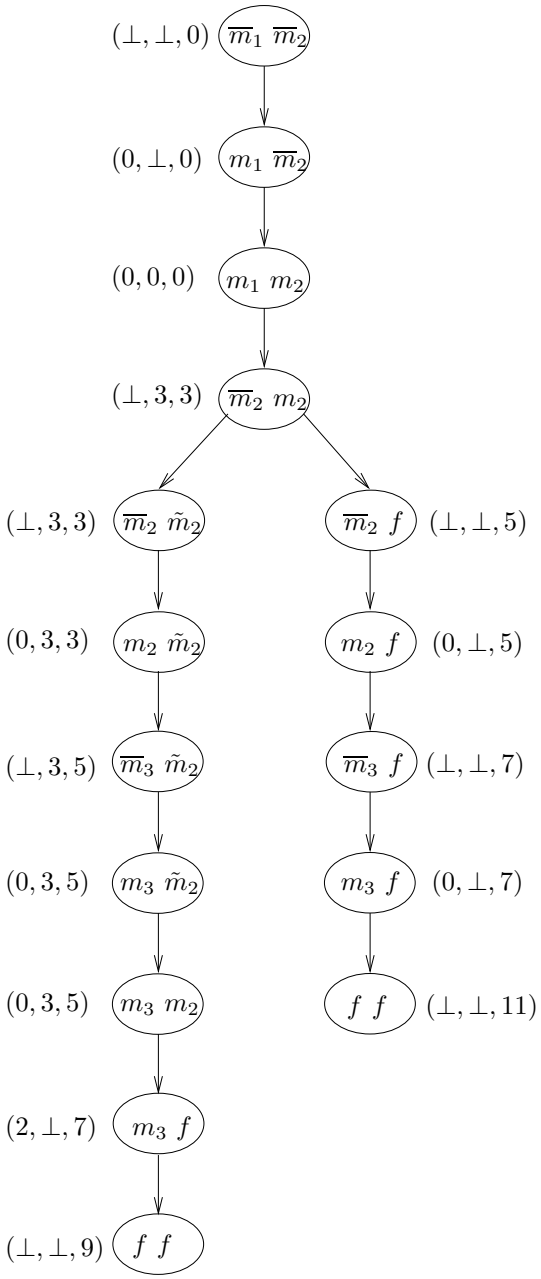


Figure 6.8: The efficient runs of the timed automaton of Figure 6.5

6.4 Experimental Results

With a best-first algorithm we were able the find optimal schedules of system with up to 8 jobs and 4 machines (12^8 discrete states and 8 clocks). We test the same heuristic proposed in the case of deterministic job shop problem on 16 difficult job-shop scheduling problems, for each of these problems we have applied our algorithms for different choices of w (it takes, on the average few minutes for each problem). In Table 6.4 we compare our best results on these problems to the most recent results reported by Le Pape and Baptiste [PB96, PB97] where the problem was solved using state-of-the-art constraint satisfaction techniques.

problem			non preempt	preemptive			
name	#j	#m	optimum	optimum	[PB96, PB97]	stopwatch	deviation
LA02	10	5	655	655	655	655	0.00 %
FT10	10	10	930	900	900	911	1.21 %
ABZ5	10	10	1234	1203	1206	1250	3.76 %
ABZ6	10	10	943	924	924	936	1.28 %
ORB1	10	10	1059	1035	1035	1093	5.31 %
ORB2	10	10	888	864	864	884	2.26 %
ORB3	10	10	1005	973	994	1013	3.95 %
ORB4	10	10	1005	980	980	1004	2.39 %
ORB5	10	10	887	849	849	887	4.28 %
LA19	10	10	842	812	812	843	3.68 %
LA20	10	15	902	871	871	904	3.65 %
LA21	10	15	1046	1033	1033	1086	4.88 %
LA24	10	15	936	909	915	972	6.48 %
LA27	10	20	1235	1235	1235	1312	5.87 %
LA37	15	15	1397	1397	1397	1466	4.71 %
LA39	15	15	1233	1221	1221	1283	4.83 %

Table 6.2: The results of our implementation on the benchmarks. Columns #j and #m indicated the number of jobs and machines, followed by the best known results for non-preemptive scheduling, the known optimum for the preemptive case, the results of Le Pape and Baptiste, followed by our results and their deviation from the optimum.

Chapter 7

Task Graph Scheduling

In this chapter we apply the methodology suggested for the job shop problem to a different problem, task graph scheduling on parallel identical machines. In this problem we have a fixed number of parallel and identical machines on which we have to execute a set of tasks linked by a set of precedence constraints represented by a task graph as in Figure 7.1. A task can be executed only if all its predecessors in this graph have completed. The job shop problem is a particular case where this graph is a set of linear chains, each chain representing the precedence relation in one job.

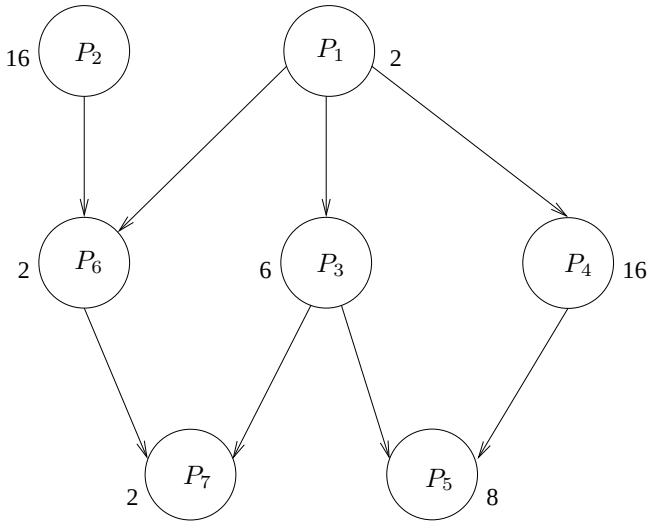


Figure 7.1: A task graph. The numbers represent task durations

Contrary to the job shop problem, a task can be executed on any idle machine and every pair of tasks can be a-priori in conflict two tasks can thus be in conflict on any machine. The significant parameter in this problem is not any more the identity of the occupied machines but the *number* of occupied machines.

7.1 The Problem

A task graph is a triple $G = (\mathcal{P}, \prec, d)$ such that $\mathcal{P} = \{P_1, \ldots, P_m\}$ is a set of m tasks, $\prec$ is a partial-order relation on $\mathcal{P}$ and $d : \mathcal{P} \to \mathbb{N}$ is a function which assigns a duration to each task.

We denote by $\Pi(P)$ the set of immediate predecessors of P. Given a set $\{M_1, \ldots, M_n\}$ of n parallel identical machines, we need to find the schedule that minimizes the total execution time and respects the following conditions:

- A task can be executed only if all its predecessors have completed.
- Each machine can process at most one task at a time.
- Tasks cannot be preempted.

If we have as many machines as we want, the optimal schedule is obtained by starting every task as soon as its predecessors terminate. In that case the length of the optimal schedule is the length of the maximal path from a minimal to a maximal element of $(\mathcal{P}, \prec$. The schedule of Figure 7.2 is an optimal schedule for the task graph of Figure 7.1 when the number of machines is unlimited. Notice that 3 machines are sufficient to construct this schedule, because no more than 3 tasks are enabled simultaneously, see Figure 7.3.

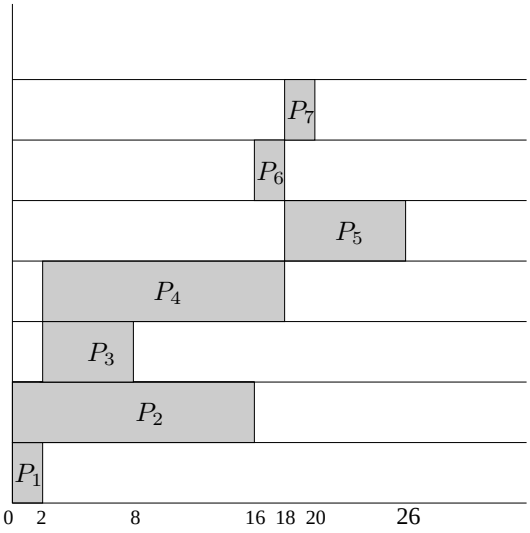


Figure 7.2: An optimal schedule of the task graph of Figure 7.1 when the number of machine is unlimited.

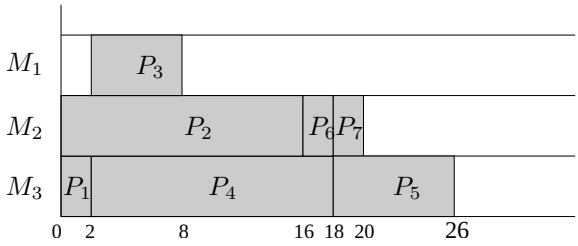


Figure 7.3: An optimal schedule of the task graph of Figure 7.1 using 3 machines.

On the other hand, if we have only 2 machines the number of enabled tasks may exceed the number of machines. We can see in schedules S_1 and S_2 of Figure 7.4 that at $t = 2$, P_2 is already

occupying M_1 where both P_3 and P_4 become enabled. In S_1 we give the remaining machine to P_3, and in S_2 we give it to P_4.

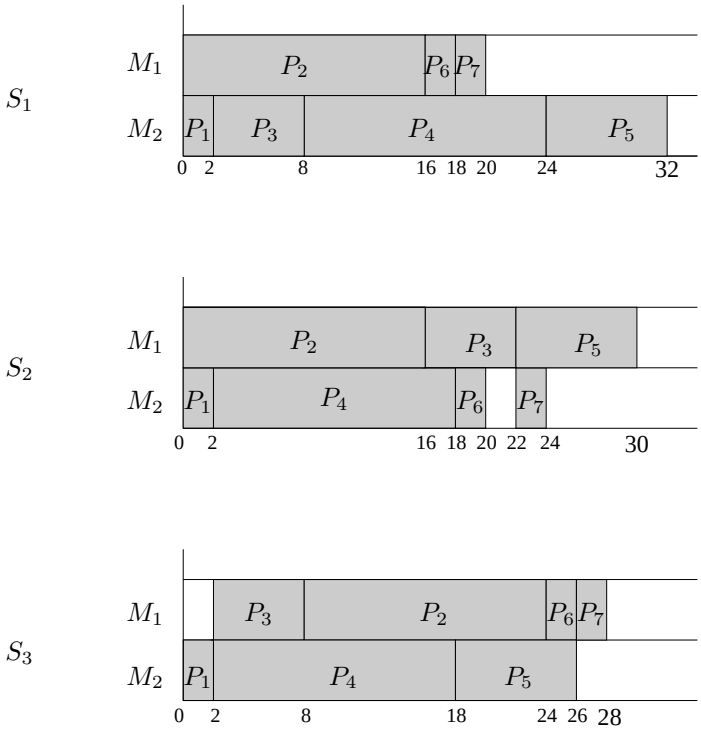


Figure 7.4: Three feasible schedules of the task graph of Figure 7.1 on 2 machines.

Unlike the case of infinitely many machine and similarly to the job-shop problem, an optimal schedule may be obtained by not executing a task as soon as it is enabled. For example, schedule S_3 achieves the optimum while not starting task P_2 as soon as possible.

7.2 Modeling with Timed Automata

Our goal is to translate this problem into a timed automaton such that every run corresponds to a feasible schedule and the shortest run gives the optimal schedule. For every task P we build a 3-state automaton with one clock c and a set of states $Q = \{\overline{p}, p, \underline{p}\}$ where $\overline{p}$ is the waiting state before the task starts, p is the active state where the task executes and $\underline{p}$ is a final state indicating that the task has terminated. The transition from $\overline{p}$ to p resets the clock and can be taken only if all the tasks in $\Pi(P)$ are in their final states. The transition from p to $\underline{p}$ is taken when $c = d(p)$. The automata for the task graph of Figure 7.1 appear in Figure 7.5.

In order to model task graph as composition of timed automata we need to modify a bit the definition of the transition relation Δ to include tuples of the form (q, ϕ, ρ, q') where ϕ is either, as before, a combination of clock inequalities, or a formula specifying states of other automata.

Definition 24 (Timed Automaton for a Task)
Let $G = (\mathcal{P}, \prec, d)$ be a task graph. For every task $P \in \mathcal{P}$ its associated timed automaton is $\mathcal{A} = (Q, \{c\}, \Delta, s, f)$ with $Q = \{p, \overline{p}, \underline{p}\}$ where the initial state is $\overline{p}$ and the final state is $\underline{p}$. The

transition relation Δ *consists of the two transitions:*

$$(\overline{p}, \bigwedge_{P \in \Pi(P)} \underline{p}', \{c\}, p)$$

and

$$(p, c = d(p), \emptyset, \underline{p})$$

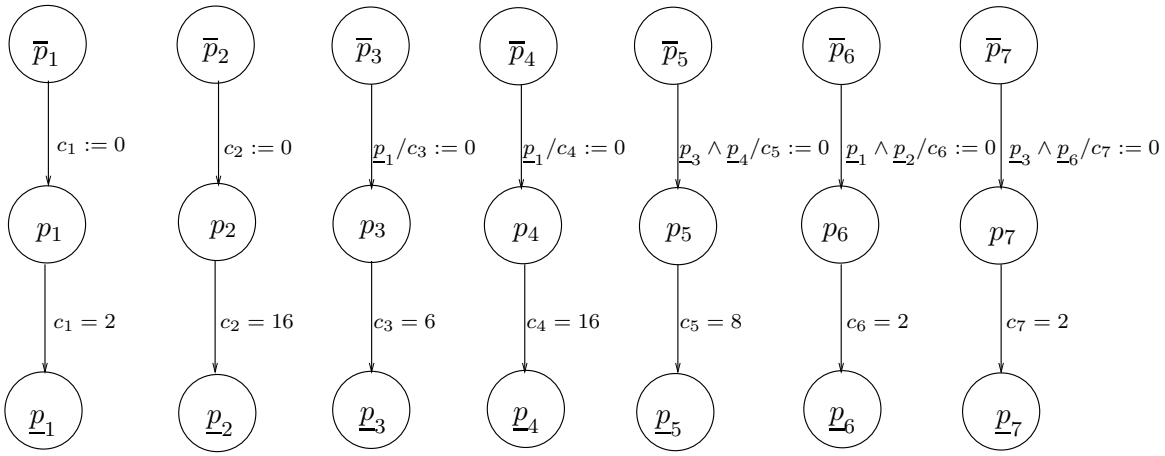


Figure 7.5: The automata for the task graph of Figure 7.1

As for the job shop problem, the global automaton representing all the feasible schedules can be obtained as a composition of the individual task automata, a composition that takes care that the number of tasks active in any global state does not exceed the number of machines. Although in terms of the number of reachable global states this automaton is as good as we can get, it has some features which make its analysis impractical and which can be improved. The global states are m-tuples where m can be very large and the number of clocks is m as well. In reality, however, even in the presence of infinitely many machines, the number of tasks that can be active simultaneously is bounded by the *width* of the task graph, the maximal number of elements incomparable with respect to $\prec$.

Definition 25 (Chain) *A chain in a partially-ordered set* $(\mathcal{P}, \prec)$ *is a subset* $\mathcal{P}'$ *of* $\mathcal{P}$ *such that for every* $P, P' \in \mathcal{P}'$ *either* $P \prec P'$ *or* $P' \prec P$.

Definition 26 (Chain Cover) *A chain covering of a partially-ordered set* $(\mathcal{P}, \prec)$ *is a set of chains* $\mathcal{H} = \{H_1, \ldots H_k\}$ *satisfying*

1. *Each* H_i *is a linearly ordered subset of* P.
2. $H_i \cap H_j = \emptyset$ *for every* $i \neq j$.
3. $\bigcup_{i \leq k} H_i = \mathcal{P}$

An example of a chain cover for our task graph appears in Figure 7.6. The structure of a chain is similar to that of a job except for the fact that tasks in one chain might depend also on the termination of tasks in other chains.

The *external predecessors* of a task $P \in H_i$ are

$$\Pi'(P) = \Pi(P) \cap (\mathcal{P} - H_i).$$

The start transition from p' to p is then enabled if for every $P' \in \Pi(P) \cup H_j$, the automaton for P' is in a state beyond p'. We denote this condition by:

$$\bigwedge_{P' \in \Pi'(P)} > p'.$$

From here we can apply the methodology developed in Chapter 5, built an automaton for every chain (Figure 7.7, compose the chain automata and apply the same search algorithms.

It is worth mentioning that chain covers are related to the width of a partial order via Dilworth's theorem [D50].

Theorem 12 (Dilworth) *The width of a partial order is equal to the minimal number of chains needed to cover it.*

Although the computation of the width and its associated cover is polynomial, we do not compute it exactly but use a fast and simple algorithm to approximate it.

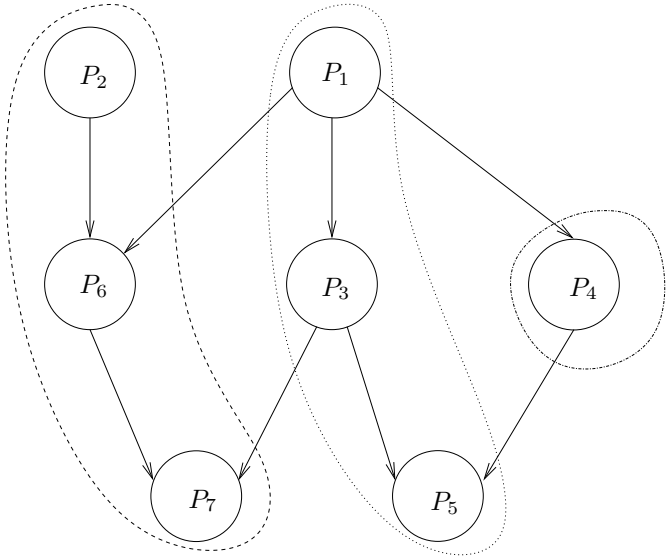


Figure 7.6: A chain covering of the task graph of Figure 7.1

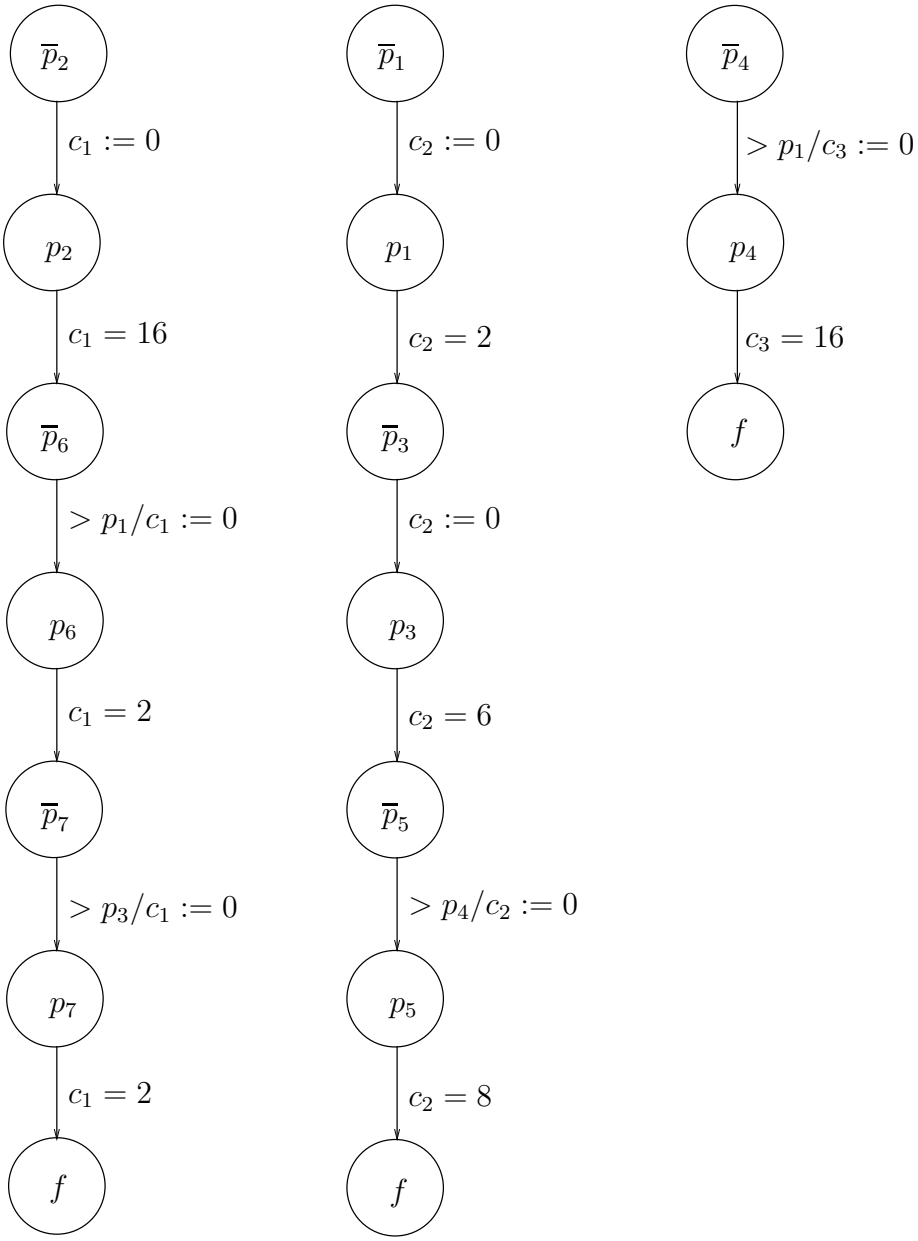


Figure 7.7: The automata for the chain cover of Figure 7.6

The timed automaton of Figure 7.8 represents a part of the timed automaton obtained by composing the automata of Figure 7.6 when there are 2 machines. This automaton has only 3 clocks (the number of chains in the cover). In the initial state, where tasks P_2, P_1 and P_4 are waiting, there are only two possible successors, to start P_2 (state $(p_2\ \overline{p}_1\ \overline{p}_4)$) or to start P_1 (state $(\overline{p}_2\ p_1\ \overline{p}_4)$). The transition to the state $(\overline{p}_2\ \overline{p}_1\ p_4)$ is disabled because task P_1 has not terminated. No start transition can be taken from $(p_2, p_3, \overline{p}_4)$ because all the machines are occupied in this state.

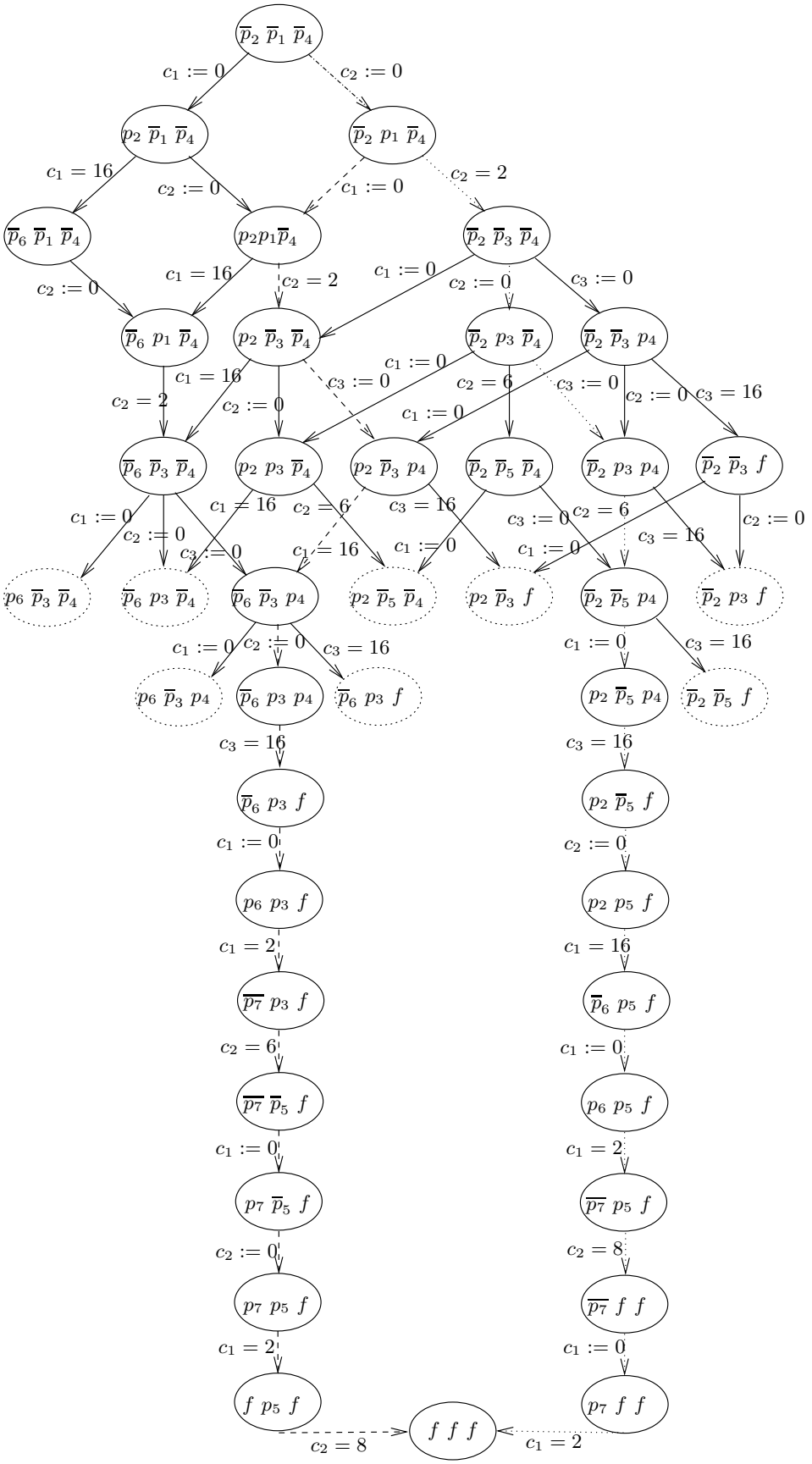


Figure 7.8: The timed automaton obtained by composing the automata of Figure 7.7 for the case of 2 machines. The two runs corresponding to the schedules S_2 and S_3 are indicated by the dashed and dotted lines, respectively.

7.3 Adding Deadlines and Release Times

Our model can be extended to include two additional feature that are often present in task graph scheduling problems ,deadlines and release times. For every task P_j a deadline $\lambda(j)$ indicates that the task must imperatively terminate before time $t = \lambda(j)$. The release time $r(j)$ indicates that the task can not be executed before time $t = r(j)$.

A feasible schedule S must respect thus two new constraints:

- $\forall P_j \in \mathcal{P}\ s(P_j) + d(j) \leq \lambda(j)$.
- $\forall P_j \in \mathcal{P}\ s(P_j) \geq r(j)$.

These features can be easily integrated into the model by making reference to the additional clock t measuring absolute time, as can be seen in Figure 7.9. This way a complete run corresponds to a feasible schedule respecting the additional constraints (a run fragment violating a deadline cannot be completed). The results concerning non-lazy schedules hold in this setting as well. It is worth mentioning that these results do not hold if we add relative deadline constraints of the form $s(P) - s(P') \leq \lambda$.

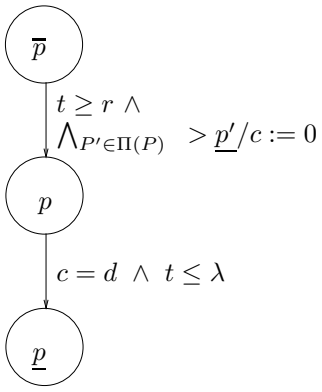


Figure 7.9: An automaton for a task with release time and deadline.

7.4 Experimental Results

To test our approach we have taken several benchmark problems from [YI99] having up to few thousands of tasks. For each of them we have applied a simple algorithm for finding a chain cover, built the automata and applied the heuristic sub-optimal shortest path algorithm. The execution time of the algorithm was around 1 minute for a problem and the results very close to the best known results as reported in Table 7.4.

name	#tasks	#cchains	# machines	[YI99]	TA
001	437	125	4	1178	1182
000	452	43	20	537	537
018	730	175	10	700	704
074	1007	66	12	891	894
021	1145	88	20	605	612
228	1187	293	8	1570	1574
071	1193	124	20	629	634
271	1348	127	12	1163	1164
237	1566	152	12	1340	1342
231	1664	101	16	t.o	1137
235	1782	218	16	t.o	1150
233	1980	207	19	1118	1121
294	2014	141	17	1257	1261
295	2168	965	18	1318	1322
292	2333	318	3	8009	8009
298	2399	303	10	2471	2473

Table 7.1: Computation of optimal schedules for benchmark problems. Our results appear in the TA column

Chapter 8

Scheduling Under Uncertainty

All models presented in the previous chapters were *deterministic* in the sense that all tasks and their durations are known in advance to the scheduler. The only non-determinism in the problem specification comes from the scheduler's decisions and once they are chosen, the system exhibits a unique run/schedule with pre-determined start times for each task. In this chapter we extend our framework to treat the more challenging problem of scheduling under uncertainty. Among the many ways to introduce uncertainty to the model, we have chosen one of the most natural ones, namely uncertainty in the duration of tasks.

8.1 The Problem

We work on a variation of the job-shop scheduling problem where the duration of each task, instead of being given, is only known to be inside an interval of the form $[l, u]$. It is the external environment that chooses each time number $d \in [l, u]$ for every task. An assignment of a number to each uncertainty interval is called an *instance*[1] of the environment.

As a running example consider two jobs

$$J^1 = (m_1, 10) \prec (m_3, [2, 4]) \prec (m_4, 5) \qquad J^2 = (m_2, [2, 8]) \prec (m_3, 7)$$

The uncertainties concern the durations of the first task of J^2 and the second task in J^1. Hence an instance is a pair $d = (d_1, d_2) \in [2, 8] \times [2, 4]$. In this example the only resource under conflict is m_3 and the order of its usage is the only decision the scheduler needs to take.

Each instance defines a deterministic scheduling problem, Figure 8.1 depicts optimal schedules for instances $(8, 4)$, $(8, 2)$ and $(4, 4)$. Of course, such an optimal schedule can only be generated by a *clairvoyant* scheduler which knows the instance in advance.

[1] Or *realization*.

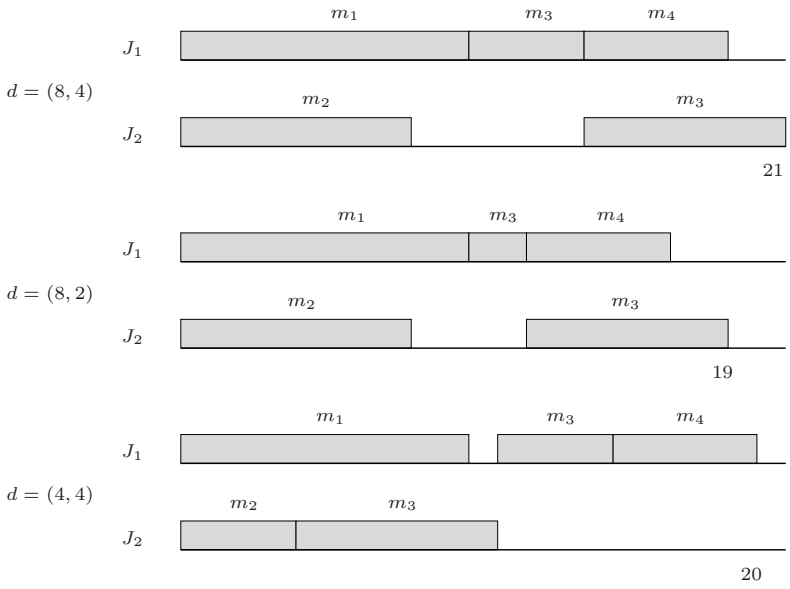


Figure 8.1: Optimal schedules for three instances. For the first two the optimum is obtained with $J^1 \prec J^2$ on m_3 while for the third — with $J^2 \prec J^1$.

If worst-case Performance is all that we care about we can do the following: find an optimal schedule for the worst instance, extract the start time for each task and stick to the schedule regardless of the actual instance. The behavior of a static schedule based on instance $(8, 4)$ is depicted in Figure 8.2, and one can see that it is rather wasteful for other instances. Intuitively we will prefer a smarter adaptive scheduler that reacts to the evolution of the environment and uses additional information revealed during the execution of the schedule. This is the essential difference between a schedule (a plan, a feed-forward controller) and a scheduling *strategy* (a reactive plan, a feedback controller). The latter is a mechanism that observes the state of the system (which tasks have terminated, which are executing) and decides accordingly what to do. In the former, since there is no uncertainty the scheduler knows exactly what will be the state at every time instant, so the strategy can be reduced to a simple assignment of start times to tasks.

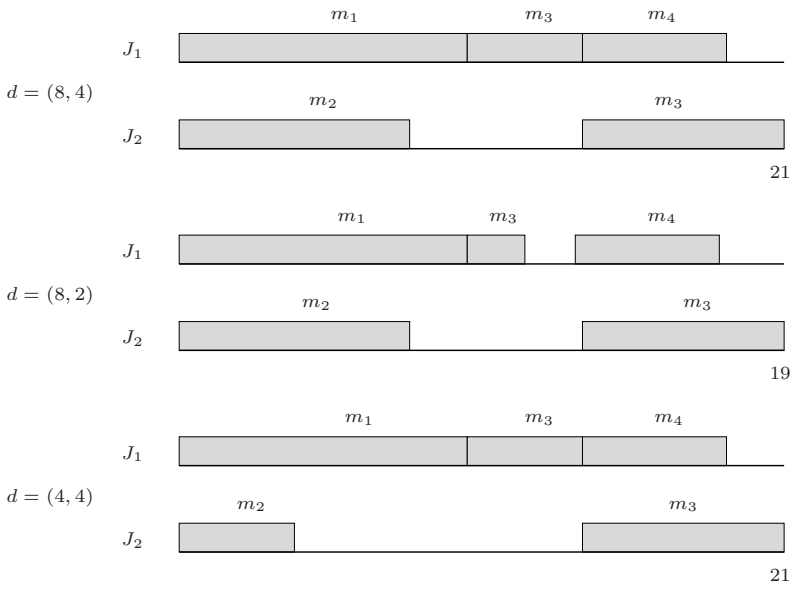


Figure 8.2: A static schedule based on the worst instance $(8, 4)$. It gives the same length for all instances.

8.2 The Hole-Filling Strategy

One of the simplest ways to be adaptive is the following. First we choose a *nominal instance* $\overline{d}$ and find a schedule S which is optimal for that instance. Rather than taking S "literally", we extract from it only the qualitative information, namely the order in which conflicting tasks utilize each resource. In our example the optimal schedule for the worst instance $(8, 4)$ is associated with the ordering $J^1 \prec J^2$ on m_3. Then, during execution, we start every task as soon as its predecessors have terminated, provided that the ordering is not violated. As Figure 8.3 shows, such a strategy is better for instances such as $(8, 2)$. It takes advantage on the earlier termination of the second task of J^1 and "shifts forward" the start times of the two tasks that follow. On the other hand, instance $(4, 4)$ cannot benefit from the early termination of m_2, because shifting m_3 of J^2 forward will violate the $J^1 \prec J^2$ ordering on m_3.

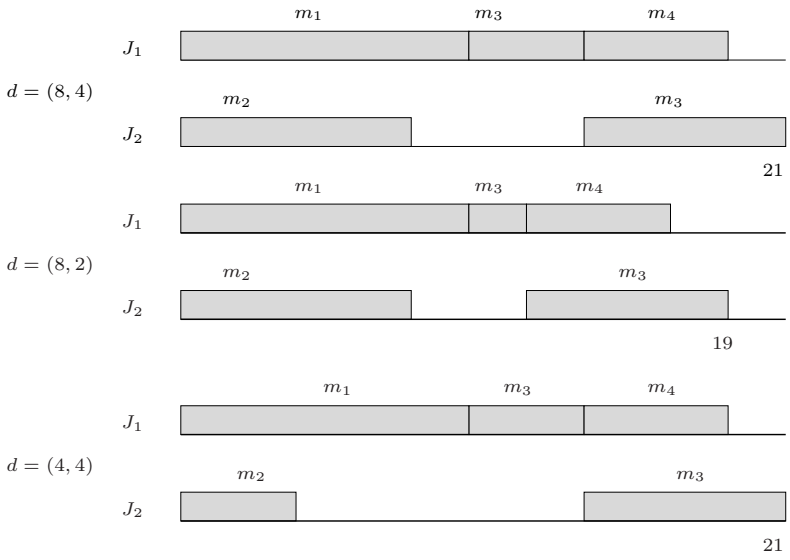


Figure 8.3: The behavior of a hole filling strategy based on instance $(8, 4)$.

Note that this "hole-filling" strategy is not restricted to the worst-case. One can use any nominal instance and then shift tasks forward or backward as needed while maintaining the order. On the other hand, a static schedule can only be based on the worst-case (a static schedule based on another nominal instance may assume a resource available at some time point, while in reality it will be occupied).

While the hole filling strategy can be shown to be optimal for all those instances whose optimal schedule has the same ordering as that of the nominal instance, it is not good for instances such as $(4, 4)$, where a more radical form of adaptiveness is required. If we look at the optimal schedules for $(8, 4)$ and $(4, 4)$ in Figure 8.1, we see that the decision whether or not to execute the second task of J^2 is done in both cases in the same qualitative state, namely m_1 is executing and m_2 has terminated. The only difference is in the time elapsed in the execution of m_1 at the decision point. Hence an adaptive scheduler should base its decisions also on such quantitative information which, in the case of timed automata models, is represented by clock values.

8.3 Adaptive Scheduling

Consider the following approach: initially we find an optimal schedule for some nominal instance. During the execution, whenever a task terminates (before or after the time we assumed it will) we reschedule the "residual" problem, assuming nominal values for the tasks that have not yet terminated. In our example, we first build an optimal schedule for $(8, 4)$. If task m_2 in J^2 terminated after 4 time we have the residual problem

$$J'_1 = (m_1, 6, !) \prec (m_3, 4) \prec (m_4, 5) \qquad J'_2 = (m_3, 7)$$

where the ! indicates that m_1 must be scheduled immediately (we assume no preemption). For this problem the optimal solution will be to start m_3 of J^2. Likewise if m_2 terminates at 8 we have

$$J'_1 = (m_1, 2, !) \prec (m_3, 4) \prec (m_4, 5) \qquad J'_2 = (m_3, 7)$$

and the optimal schedule consists of waiting for the termination of m_1 and then starting m_3 of J^1. The property of the schedules obtained this way, is that at any moment in the execution they are optimal with respect to the nominal assumption concerning the *future*.[2]

This approach involves a lot of *online* computation, namely solving a new scheduling problem each time a task terminates. The alternative approach that we propose in this chapter is based on expressing the scheduling problem using timed automata and synthesizing a control strategy *off-line*. In this framework [AMPS98, AM99, AGP99] a strategy is a function from states and clock valuations to controller actions (in this case, starting tasks). After computing such a strategy and representing it properly, the execution of the schedule may proceed while keeping track of the state of the corresponding automaton. Whenever a task terminates, the optimal action is quickly computed from the strategy look-up table and the results are identical to those obtained via online re-scheduling.

We will use notations similar to the partial-order precedence used in Chapter 7 although our examples consist of linearly ordered jobs.

Definition 27 (Uncertain Job-Shop Specification)
An uncertain job-shop specification is $\mathcal{J} = (P, M, \prec, \mu, D, U)$ *where* P *is a finite number of tasks,* M *is a finite set of machines,* $\prec$ *is a partial-order precedence relation on tasks,* $\mu : P \to M$ *assigns tasks to machines,* $D : P \to \mathbb{N} \times \mathbb{N}$ *assigns an integer-bounded interval to each task and* U *is a subset of immediate tasks consisting of some* $\prec$*-minimal elements.*

The set U is typically empty in the initial definition of the problem and we need it to define residual problems. We use D^l and D^u to denote the projection of D on the lower- and upper-bounds of the interval, respectively. The set $\Pi(p) = \{p' : p' \prec p\}$ denotes all the predecessors of p, namely the tasks that need to terminate before p starts. In the standard job-shop scheduling problem, $\prec$ decomposes into a disjoint union of chains (linear orders) called jobs.

An *instance* of the environment is any function $d : P \to \mathbb{R}_+$, such that $d(p) \in D(p)$ for every $p \in P$. The set of instances admits a natural partial-order relation: $d \leq d'$ if $d(p) \leq d'(p)$ for every $p \in P$. Any environment instance induces naturally a deterministic instance of $\mathcal{J}$, denoted by $\mathcal{J}(d)$, which is a classical job-shop scheduling problem. The worst-case is defined by the maximal instance $\overline{d}(p) = D^u(p)$ for every p.

Definition 28 (Schedule) *Let* $\mathcal{J} = (P, M, \prec, \mu, D, U)$ *be an uncertain job-shop specification and let* $\mathcal{J}(d)$ *be a deterministic instance. A feasible schedule for* $\mathcal{J}(d)$ *is a function* $s : P \to \mathbb{R}_+$*, where* $s(p)$ *defines the start time of task* p*, satisfying:*

1. *Precedence: For every* p*,* $s(p) \geq \max_{p' \in \Pi(p)}(s(p') + d(p'))$*.*
2. *Mutual exclusion: For every* p, p' *such that* $\mu(p) = \mu(p')$

$$[s(p), s(p) + d(p)] \cap [s(p'), s(p') + d(p')] = \emptyset.$$

3. *Immediacy: For every* $p \in U$*,* $s(p) = 0$*.*

In order to be adaptive we need a *scheduling strategy*, i.e. a rule that may induce a different schedule for every d. However, this definition is not simple because we need to restrict ourselves to *causal* strategies, strategies that can base their decisions only on information *available at the time they are made*. In our case, the value of $d(p)$ is revealed only when p terminates.

[2] A similar idea is used in *model-predictive control* where at each time actions at the current "real" state are re-optimized while assuming some "nominal" prediction of the future.

Definition 29 (State of Schedule) *A state of a schedule is $S = (P^f, P^a, c, P^e)$ such that P^f is a downward-closed subset of $(P, \prec)$ indicating the tasks that have terminated, P^a is a set of active tasks currently being executed, $c : P^a \to \mathbb{R}_+$ is a function such that $c(p)$ indicates the time elapsed since the activation of p and P^e is the set of enabled tasks consisting of those whose predecessors are in P^f. The set of all possible states is denoted by $\mathcal{S}$.*

Definition 30 (Scheduling Strategy) *A (state-based) scheduling strategy is a function $\sigma : \mathcal{S} \to P \cup \{\bot\}$ such that for every $S = (P^f, P^a, c, P^e)$, $\sigma(S) = p \in (P^e \cup \{\bot\})$ and for every $p' \in P^a$, $\mu(p) \neq \mu(p')$.*

In other words the strategy decides at each state whether to do nothing and let time pass ($\bot$) or to choose an enabled task, not being in conflict with any active task, and start executing it. An operational definition of the interaction between a strategy and an instance will be given later using timed automata, but intuitively one can see that the evolution of the state of a schedule consists of two types of transitions: *uncontrolled* transitions where an active task p terminates after $d(p)$ time and moves from P^a to P^f, leading possibly to adding new tasks to P^e, and a decision of the scheduler to start an enabled task. The combination of a strategy and an instance yields a unique schedule $s(d, \sigma)$ and we say that a state is (d, σ)-reachable if it occurs in $s(d, \sigma)$.

Next we formalize the notion of a residual problem, namely a specification of what remains to be done in an intermediate state of the execution.

Definition 31 (Residual Problem) *Let $\mathcal{J} = (P, M, \prec, \mu, D, U)$ and let $S = (P^f, P^a, c, P^e)$ be a state. The residual problem starting from S is $\mathcal{J}_S = (P - P^f, M, \prec', \mu', D', P^a)$ where $\prec'$ and μ' are, respectively, the restrictions of $\prec$ and μ, to $P - P^f$ and D' is constructed from D by letting*

$$D'(p) = \begin{cases} D(p) \dot{-} c(p) & \text{if } p \in P^a \\ D(p) & \text{otherwise} \end{cases}$$

Definition 32 ($\overline{d}$-Future-Optimal Strategies) *Let $\overline{d}$ be an instance. A strategy σ is* $\overline{d}$-future optimal *if for every instance d and from every (σ, d)-reachable state S, it produces the optimal schedule for $\mathcal{J}_S(\overline{d})$.*

This is exactly the property of the online re-scheduling approach described informally in the previous section.

8.4 Modeling with Timed Automata

For each task we construct a timed automaton $\mathcal{A}_D$ that captures all instances of the task: this automaton can stay in an active state p as long as $c \leq u$ and can leave p as soon as as $c \geq l$. It represents the possible behaviors of the task *in isolation*, i.e. ignoring precedence and resource constraints. The transition from a waiting state $\overline{p}$ to p is triggered by a decision of the scheduler, while the time of the transition from p to $\underline{p}$ depends on the instance. For a given instance d we have the automaton $\mathcal{A}_d$ of Figure 8.4-(b) where this transition happens after exactly d time. The automaton $\mathcal{A}_{D,d}$ of Figure 8.4-(c) will be used later for computing d-future optimal strategies: it can terminate as soon as $c \geq d$ but can stay in p until $c = u$.

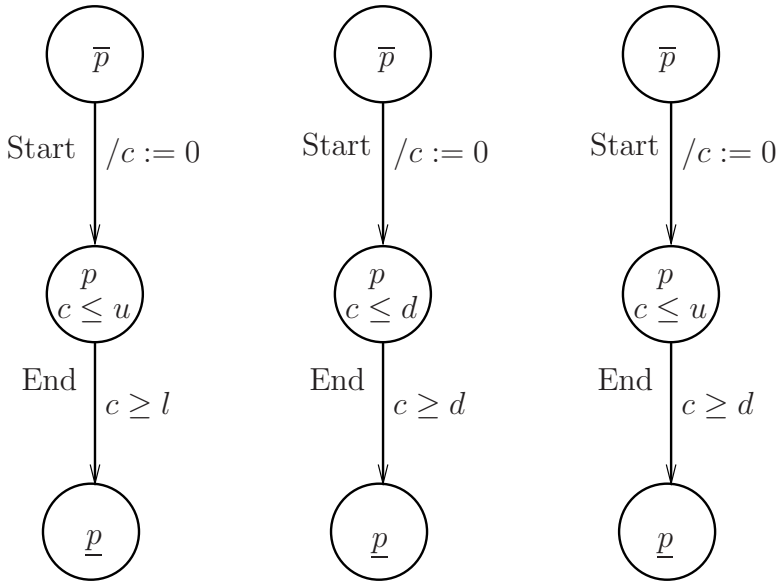


Figure 8.4: (a) The generic automaton $\mathcal{A}_D$ for a task p such that $D(p) = [l, u]$. (b) The automaton $\mathcal{A}_d$ for a deterministic instance d. (c) The automaton $\mathcal{A}_{D,d}$ for computing d-future optimal strategies. Staying conditions for $\overline{p}$ and $\underline{p}$ are **true** and omitted from the figure.

The timed automaton for the whole job-shop specification corresponding to the example appears in Figure 8.5. The automaton can be viewed as specifying a *game* between the scheduler and the environment. The environment can decide whether or not to take an "end" transition and terminate an active task and the scheduler can decide whether or not to take some enabled "start" transition. A strategy is a function that maps any configuration of the automaton either into one of its transition successors or to the waiting "action". For example, at $(m_1, \overline{m}_3)$ there is a choice between moving to (m_1, m_3) by giving m_3 to J^2 or waiting until J^1 terminates m_1 and letting the environment take the automaton to $(\overline{m}_3, \overline{m}_3)$, from where the conflict concerning m_3 can be resolved in either of the two possible ways.

A strategy is d-future optimal if from every configuration reachable in $\mathcal{A}_{D,d}$ it gives the shortest path to the final state. In the next section we use a simplified form of the definitions and the algorithm of [AM99] to find such strategies.

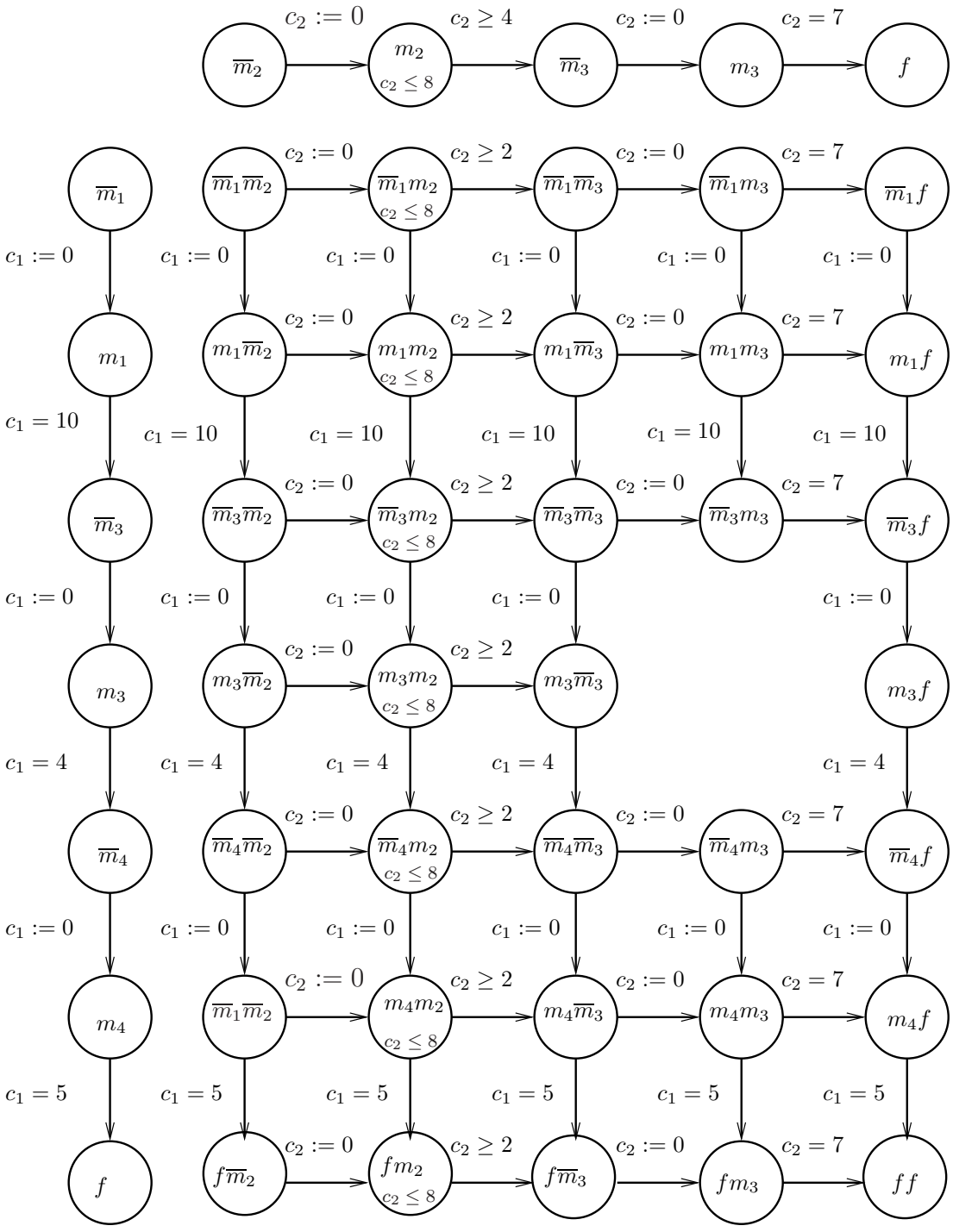


Figure 8.5: The global automaton for the job-shop specification. The automata on the left and upper parts of the figure are the partial compositions of the automata for the tasks of J^1 and J^2, respectively.

8.5 Optimal Strategies for Timed Automata

Let $\mathcal{J}$ be a job-shop specification and let $\mathcal{A}_{D,d} = (Q, C, s, f, I, \Delta)$ be the automaton corresponding to an instance d, that is, "end" transitions are guarded by conditions of the form $c_i \geq d(p_i)$. Let $h : Q \times \mathcal{H} \to \mathbb{R}_+$ be a function such that $h(q, \mathbf{v})$ is the length of the minimal run from $(q, \mathbf{v})$ to f, assuming that all uncontrolled future transitions will be taken according to d. This function admits the following recursive backward definition:

$$h(f, \mathbf{v}) = 0$$

$$h(q, \mathbf{v}) = \min\{t + h(q', \mathbf{v}') : (q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1}) \xrightarrow{0} (q', \mathbf{v}')\}.$$

In other words, $h(q, \mathbf{v})$ is the minimum over all successors q' of q of the time it takes from $(q, \mathbf{v})$ to satisfy the transition guard to q' plus the time to reach f from the resulting configuration $(q', \mathbf{v}')$. In [AM99] it has been shown that h ranges over a class of "nice" functions, closely related to zones, and that this class is well-founded and hence the computation of h terminates even for automata with cycles, a fact that we do not need here as h is computed in one sweep through all paths from the final to the initial state.

Let us illustrate the computation of h on our example. We write the function in the form $h(q^1, q^2, c_1, c_2)$ and use $\perp$ to denote cases where the value of the corresponding clock is irrelevant (its task is not active). We start with

$$\begin{array}{l} h(f, f, \perp, \perp) = 0 \\ h(m_4, f, c_1, \perp) = 5 \dot{-} c_1 \\ h(f, m_3, \perp, c_2) = 7 \dot{-} c_2 \end{array}$$

This is because the time to reach (f, f) from (m_4, f) is the time it takes to satisfy the guard $c_1 = 5$, etc. The value of h at (m_4, m_3) depends on the values of both clocks which determine what will terminate before, m_4 or m_3 and whether the shorter path goes via (m_4, f) or (f, m_3).

$$\begin{aligned} h(m_4, m_3, c_1, c_2) &= \min \left\{ \begin{array}{l} 7 \dot{-} c_2 + h(m_4, f, c_1 + (7 \dot{-} c_2), \perp), \\ 5 \dot{-} c_1 + h(f, m_3, \perp, c_2 + (5 \dot{-} c_1)) \end{array} \right\} \\ &= \min\{5 \dot{-} c_1, 7 \dot{-} c_2\} \\ &= \left\{ \begin{array}{ll} 5 \dot{-} c_1 & \text{if } c_2 \dot{-} c_1 \geq 2 \\ 7 \dot{-} c_2 & \text{if } c_2 \dot{-} c_1 \leq 2 \end{array} \right. \end{aligned}$$

Note that in this state the outgoing transitions are both uncontrolled "end" transition and no decision of the scheduler is required.

This procedure goes higher and higher in the graph, computing h for the whole state-space $Q \times \mathcal{H}$. In particular, for state $(m_1, \overline{m}_3)$ where we need to decide whether to start m_3 of J^2 or to wait, we obtain:

$$\begin{aligned} h(m_1, \overline{m}_3, c_1, \perp) &= \min\{16, 21 \dot{-} c_1\} \\ &= \left\{ \begin{array}{ll} 16 & \text{if } c_1 \leq 5 \\ 21 \dot{-} c_1 & \text{if } c_1 \geq 5 \end{array} \right. \end{aligned}$$

The extraction of a strategy from h is straightforward: if the optimum of h at $(q, \mathbf{v})$ is obtained via a controlled transition to q' we let $\sigma(q, \mathbf{v}) = q'$ otherwise if it is obtained via an uncontrolled transition we let $\sigma(q, \mathbf{v}) = \perp$. For $(m_1, \overline{m}_3)$ the optimal strategy is

$$\sigma(m_1, \overline{m}_3, c_1, \perp) = \left\{ \begin{array}{ll} (m_1, m_3) & \text{if } c_1 \leq 5 \\ \perp & \text{if } c_1 \geq 5 \end{array} \right.$$

meaning that if m_1 is used for less than 5 time units we give m_3 to J^2 and if it has been used for more than 5 time units we wait until it terminates and give machine m_2 to J^1. Note that if we assume that J^1 and J^2 started their first tasks simultaneously, the value of c_1 upon entering $(m_1, \overline{m}_3)$ is exactly the duration of m_2 in the instance.

The results of Chapter **??** concerning "non-lazy" schedules imply that optimal strategies have the additional property that if $\sigma(q, \mathbf{v}) = \bot$ then $\sigma(q, \mathbf{v}') = \bot$ for every $\mathbf{v}' \geq \mathbf{v}$. In other words, if an enabled controlled transition gives the optimum it should be taken as soon as possible. This fact will be used later in the implementation of the strategy.

Theorem 13 (Computability of Optimal Strategies)
Given an uncertain job-shop specification and an instance $\overline{d}$ it is possible to compute a $\overline{d}$-future optimal scheduling strategy.

This result is a special case of the result of [AM99].

8.5.1 Implementation

Existing algorithms for timed automata work on sets, not on functions, and in order to apply them to the computation of h we use the following construction. Let $\mathcal{A}'$ be an auxiliary automaton augmented with a clock representing absolute time. Clearly, if $(q, \mathbf{v}, t)$ is reachable in $\mathcal{A}'$ from the initial state $(s, \mathbf{0}, 0)$ then $(q, \mathbf{v})$ is reachable in $\mathcal{A}$ in time t.

Let Θ be a positive integer larger then the longest path in the automaton. Starting from $(f, \bot, \Theta)$ and doing backward reachability we can construct a relational representation of h. More precisely, if $(q, \mathbf{v}, t)$ is backward reachable in the extended timed automaton $\mathcal{A}'$ from $(f, \{\bot\}, \Theta)$ then f is forward reachable in $\mathcal{A}$ from $(q, \mathbf{v})$ within $\Theta - t$ time.

Applying the standard backward reachability algorithm for timed automata we compute the set R of all backward-reachable symbolic states. In order to be able to extract strategies we store tuples of the form (q, Z, q') such that Z is a zone of $\mathcal{A}'$ and q' is the successor of q from which (q, Z) was reached backwards.

The set R gives sufficient information for implementing the strategy. Whenever a transition to $(q, \mathbf{v})$ is done during the execution we look at all the symbolic states with discrete state q and find

$$h(q, \mathbf{v}) = \min\{\Theta - t : (\mathbf{v}, t) \in Z \wedge (q, Z, q') \in R\}.$$

If q' is a successor via a controlled transition, we move to q', otherwise we wait until a task terminates and an uncontrolled transition is taken. Non-laziness guarantees that we need not revise a decision to wait until the next transition.

8.6 Experimental Results

We have implemented the algorithm using the zone library of Kronos [BDM+98], as well as the hole-filling strategy and the algorithm for the exponential distribution. As a benchmark we took the following problem with 4 jobs and 6 machines:

$$\begin{array}{ll}
J_1: & (m_2, [\,4, 10]) \prec (m_4, [\,1,\ 7]) \prec (m_3, [28, 40]) \prec (m_1, [\,7, 15]) \prec (m_5, [\,6, 25]) \prec (m_6, [45, 63]) \\
J_2: & (m_5, [14, 25]) \prec (m_1, [34, 46]) \prec (m_2, [\,2, 27]) \prec (m_4, [\,9, 14]) \prec (m_6, [14, 29]) \prec (m_3, [32, 48]) \\
J_3: & (m_4, [47, 55]) \prec (m_6, [32, 46]) \prec (m_1, [\,4, 12]) \prec (m_5, [\,1, 14]) \prec (m_2, [\,5, 16]) \prec (m_3, [\,4,\ 9]) \\
J_4: & (m_6, [54, 72]) \prec (m_2, [21, 36]) \prec (m_3, [\,1,\ 8]) \prec (m_4, [22, 37]) \prec (m_1, [\,7, 18]) \prec (m_5, [\,4, 18])
\end{array}$$

The static worst-case schedule for this problem is 210. We have applied Algorithm 1 to find d-future optimal strategies based on three instances that correspond, respectively, to "optimistic",

"realistic" and "pessimistic" predictions. For every p such that $D(p) = [l, u]$ they are defined as

$$d_{min}(p) = l \quad d_{avg}(p) = (l + u)/2 \quad d_{max}(p) = u.$$

We have generated random instances and compared the results of the abovementioned strategies with an optimal clairvoyant scheduler[3] that knows d in advance, and a static worst-case scheduler. The first table compares the performance on 30 instances where durations are drawn uniformly from the $[l, u]$ intervals. As it turns out that the pessimistic adaptive strategy, based on d_{max}, is very good and robust. It gives schedules that, on the average, are only 2.39% longer than those produced by a clairvoyant scheduler. For comparison, the static worst-case strategy deviates from the optimum by an average of 16.18%. On the other hand the realistic and optimistic strategies are usually inferior to the pesimistic one and in some instances they are even worse than the static schedule. This can be explained by the fact that schedules that rely on the minimal prediction are almost always not executed as planned. The hole-filling strategy based on worst-case prediction achieves good performance (3.73% longer than the optimum) with a much more modest computational effort (the results of hole-filling based on optimisitc and realistic predictions are bad and are not shown in the table).

[3]In the domain called *online algorithms* it is common to compare the performance of algorithms that receive their inputs progressively to a clairvoyant algorithm and the relation between their performances is called the *competitive ratio*.

Inst	Opt	Static	%	Max	%	Avg	%	Min	%	Hole	%
1	172	204	18.60	172	0.00	186	8.13	205	19.18	174	1.16
2	193	210	8.80	193	0.00	193	0.00	193	0.00	210	8.81
3	157	203	29.29	172	9.55	178	13.37	189	20.38	157	0.00
4	175	208	16.00	181	3.43	181	3.42	194	10.85	175	0.00
5	177	199	12.42	177	0.00	198	11.86	196	10.73	192	8.47
6	186	209	12.36	189	1.16	192	3.22	189	1.61	186	0.00
7	176	203	15.34	177	0.57	180	2.27	197	11.93	184	4.55
8	176	203	15.34	186	5.68	209	18.75	204	15.90	186	5.68
9	180	206	14.44	180	0.00	186	3.33	195	8.33	181	0.56
10	167	204	22.15	171	2.40	170	1.79	183	9.58	167	0.00
11	202	206	1.98	202	0.00	202	0.00	203	0.49	202	0.00
12	166	202	6.87	166	0.00	172	3.61	197	18.67	175	5.42
13	189	202	6.87	189	0.00	189	0.00	221	16.93	199	5.29
14	176	199	13.06	176	0.00	184	4.54	192	9.09	185	5.11
15	180	204	13.33	180	0.00	185	2.77	192	6.66	189	5.00
16	167	204	22.15	171	2.40	175	4.79	178	6.58	177	5.99
17	178	204	14.60	180	1.12	187	5.05	201	12.92	188	5.62
18	175	202	15.42	182	4.00	184	5.14	204	17.24	182	4.00
19	174	202	16.09	174	0.00	181	4.02	191	9.77	175	0.57
20	176	201	14.20	180	2.27	183	9.97	192	9.09	190	5.68
21	170	199	17.05	170	0.00	187	10.00	182	70.5	171	0.59
22	167	202	20.95	168	0.60	174	4.19	183	9.58	180	1.80
23	185	210	13.51	185	0.00	185	0.00	200	8.10	189	2.16
24	170	204	20.00	191	12.35	177	4.11	176	3.52	187	10.00
25	158	203	28.48	163	3.16	168	6.32	185	17.08	165	4.43
26	171	204	19.29	193	12.87	193	12.86	200	11.73	171	0.00
27	179	199	11.17	179	0.00	193	7.82	210	17.31	179	0.00
28	174	203	16.66	180	3.45	182	4.59	202	16.09	174	0.00
29	162	201	24.07	170	4.94	171	5.55	183	12.96	172	6.16
30	172	200	16.27	179	4.07	193	12.02	183	6.39	184	6.98
Avg	175	203.33	16.18	179.19	2.39	184.60	5.48	191.93	9.67	181.53	3.73

Chapter 9

Conclusions

In this thesis we have laid the foundations for an automaton-based scheduling methodology. As a first step we have attacked problems such as the job-shop problem that can be solved using existing techniques, in order to see if the performance of timed automata technology is acceptable. As it turned out, the standard zone-based algorithms for timed automata were too heavy and this led us to the discovery of non-lazy schedules and runs. Using points instead of zones, we could compete with other techniques and this insight may be useful also for standard verification of timed automata.

The next step was to consider preemption, where the cyclic nature of the automaton poses problems also for other techniques. If the number of preemptions is not bounded, there is no bound on the number of variables in the corresponding constrained optimization problem. This led us to the rediscovery of "Jackson schedule" from the 50s, what we call "efficient". With this result, a point based search algorithm can be applied, leading to competetive performance. It also shows that the undecidability results for stopwatch automata are not always relevant.

The generalization to the task graph problem was rather straightforward, the only significant new feature was the decomposition of the partial order to form a chain cover, and the adaptation of the mutual exclusion condition to identical machines. Again the performance of our algorithm was not worse than the state-of-the-art in the domain.

The treatment of uncertainty in Chapter 8 was supposed to be the first domain where the advantages of a state-based approach like ours become visible. And indeed, the definition and computation of adaptive strategies would be very hard to perform without the insight coming from the automaton model. However we consider the results so far only as a partial success because the backward reachability algorithm in its current form has to explore all the state-space (unlike a best-first search used in forward reachability) and use the expensive zone technology. This implies that the size of problems that can be treated is much smaller than for the deterministic case. Much more work is needed to extend the scope of scheduling strategy synthesis. On the other hand, a less adaptive strategy such as hole filling can be easily implemented on top of our solution of the deterministic problem.

For future work we envisage too major research directions, one concerned with improving the performance and the other in extending the models to treat more complex scheduling situations. The first direction include:

1. New search heuristics for deterministic problems.

2. New techniques for "compositional" scheduling. The basic idea is that jobs can be grouped into subsets and an optimal schedule can be found for each subset. Then, each obtained schedule for a subset can be seen as a job by itself, and the composition of these "super jobs" defines a new scheduling problem. Clearly this technique is not guaranteed to give an

optimal solution but it may suggest a practical approach for treating a very large number of jobs..

3. Combination of forward and backward reachability in strategy synthesis. The hardness of strategy synthesis is that a-priori a strategy should be computed for every reachable configuration, but the definition of a reachable configuration depends on the strategy. Consequently our current algorithm computes the strategy for all reachable configuration under any strategy. Is is clear, however, that some strategies (especially very lazy ones) will never be chosen. If, using forward analysis, we can restrict the set of states for which we need to compute a strategy, we can restrict significantly the computation time. Moreover, if we accept sub-optimal strategies, we can refrain from computing the strategy even for some reachable states and use default actions if these states are reached in the actual schedule.

Among the many possible extensions of the scheduling problems we mention the following::

1. Job-shop problems where some steps can be executed on different machines with different speeds.

2. Problems where the identity of the next step to be executed depends on the result of the current step. Such situation occur, for example, in computer programs.

3. Problems where some of the structure of the tasks depends also on the previous choices of the scheduler. For example, if machines are distributed geographically, different choices of machines will imply different steps of transportation. Likewise, in task graph scheduling, communication should be added between tasks that do not execute on the same machine.

4. Problems with more complex timing constraints for which the laziness results do not hold. For example, problem with relative deadlines or synchronization constraints in task graph scheduling.

We believe that, regardless of the current applicability of our techniques to real-life industrial-size problems, the framework developed in this thesis adds a new perspective from which scheduling problems can be viewed, understood and solved.

Bibliography

[AM02] Y. Abdeddaim and O. Maler, Preemptive Job-Shop Scheduling using Stopwatch Automata, in J.Katoen and P.Stevens (Eds.), TACAS'02, 113-126, LNCS 2280, Springer 2002.

[AM01] Y. Abdeddaïm and O. Maler, Job-Shop Scheduling using Timed Automata in G. Berry, H. Comon and A. Finkel (Eds.), *Proc. CAV'01*, 478-492, LNCS 2102, Springer 2001.

[ATP01] R. Alur, S. La Torre and G.J. Pappas, Optimal Paths in Weighted Timed Automata, *Proc. HSCC'01*, 49-64, LNCS 2034, Springer 2001.

[AGP99] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis and S. Yovine, A Framework for Scheduler Synthesis. *Proc. RTSS'99*, 154-163, IEEE, 1999.

[AM99] E. Asarin and O. Maler, As Soon as Possible: Time Optimal Control for Timed Automata, *Proc. HSCC'99*, 19-30, LNCS 1569, Springer, 1999.

[AMPS98] E. Asarin, O. Maler, A. Pnueli and J. Sifakis, Controller Synthesis for Timed Automata, *Proc. IFAC Symposium on System Structure and Control*, 469-474, Elsevier, 1998.

[AMP95] E. Asarin, O. Maler and A. Pnueli, Symbolic Controller Synthesis for Discrete and Timed Systems, *Hybrid Systems II*, LNCS 999, Springer, 1995.

[AD94] R. Alur and D.L. Dill, A Theory of Timed Automata, *Theoretical Computer Science* 126, 183-235, 1994.

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill, Model Checking in Dense Real Time, *Information and Computation* 104, 2-34, 1993.

[ABC91] D. Applegate, and W. Cook, A Computational Study of the Job-Shop Scheduling Problem, ORSA 149-156, Journal on Computing, Spring, 3(2), 1991.

[ABZ88] J. Adams, E. Balas, and D. Zawack, The Shifting Bottleneck Procedure for Job-Shop Scheduling, 391-401, Management Science, March, 34(3), 1988.

[A67] S .Ashour, A Decomposition Approach for the Machine Scheduling Problem, 109-122,International Journal of Production Research 6(2), 1967.

[BFH+01a] G. Behrmann, A. Fehnker, T.S. Hune, K.G. Larsen, P. Pettersson and J. Romijn, Efficient Guiding Towards Cost-Optimality in UPPAAL, *Proc. TACAS 2001*, 174-188, LNCS 2031, Springer, 2001.

[BFH+01b] G. Behrmann, A. Fehnker T.S. Hune, K.G. Larsen, P. Pettersson, J. Romijn and F.W. Vaandrager, Minimum-Cost Reachability for Linearly Priced Timed Automata, *Proc. HSCC'01*, 147-161, LNCS 2034, Springer 2001.

[BS99] R. Boel and G. Stremersch, VHS case study 5: Modelling and Verification of Scheduling for Steel Plant at SIDMAR, Draft, 1999.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, Kronos: a Model-Checking Tool for Real-Time Systems, *Proc. CAV'98*, LNCS 1427, Springer, 1998.

[BB96] E.A. Boyd, and R. Burlingame, A Parallel Algorithm for Solving Difficult Job-Shop Scheduling Problems, Operations Research Working Paper, Department of Industrial Engineering, Texas A & M University, College Station, Texas 77843-3131, USA, 1996.

[BPN95] P. Baptiste, C. Le Pape, and W.P.M Nuijten, Constraint-Based Optimization and Approximation for Job-Shop Scheduling, AAAI-SIGMAN Workshop on Intelligent Manufacturing Systems, 14 th International Joint Conference on Artificial Intelligence, Montréal, Québec, Canada, Aug 19, pp. 5-16, 1995.

[BJS94] P. Brucker, B. Jurisch, and B. Sievers, A Branch and Bound Algorithm for the Job-Shop Scheduling Problem, 109-127, Discrete Applied Mathematics, vol 49, 1994 .

[B69] E. Balas, Machine Scheduling via Disjunctive Graphs: An Implicit Enumeration Algorithm, 941-957, Operations Research, vol 17, 1969 .

[B59] E.H. Bowman, The Schedule-Sequencing Problem, 621-624, Operations Research, vol 7, 1959.

[CL00] F. Cassez and K.G. Larsen, On the Impressive Power of Stopwatches, in C. Palamidessi (Ed.) *Proc. CONCUR'2000*, 138-152, LNCS 1877, Springer, 2000.

[CCM+94] S. Campos, E. Clarke, W. Marrero, M. Minea and H. Hiraishi, Computing Quantitative Characteristics of Finite-state Real-time Systems, *Proc. RTSS'94*, IEEE, 1994.

[CY94] Y. Caseau and F. Laburthe, Improved CLP Scheduling with Task Intervals, in Van Hentenryck, P. (ed) ICLP94 Proceedings of the Eleventh International Conference on Logic Programming, MIT Press, 1994.

[CPP92] C. Chu, M.C. Portmann, and J.M. Proth, A Splitting-Up Approach to Simplify Job-Shop Scheduling Problems, 859-870, International Journal of Production Research 30(4), 1992.

[C92] K. Cerans, *Algorithmic Problems in Analysis of Real Time System Specifications*, Ph.D. thesis, University of Latvia, Riga, 1992.

[CHR91] Z. Chaochen, C.A.R. Hoare and A.P. Ravn, A Calculus of Durations, *Information Processing Letters* 40, 269-276, 1991.

[CY91] C. Courcoubetis and M. Yannakakis, Minimum and Maximum Delay Problems in Real-time Systems, *Proc. CAV'91*, LNCS 575, 399-409, Springer, 1991.

[CP90] J. Carlier and E. Pinson, A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem, *Annals of Operations Research* 26, 1990.

[C82] J .Carlier, The One-Machine Sequencing Problem, 42-47, European Journal of Operational Research, vol 11, 1982.

[DY96] C. Daws and S. Yovine, Reducing the Number of Clock Variables of Timed Automata, *Proc. RTSS'96*, 73-81, IEEE, 1996.

[DR94] F. Della Croce, R. Tadei, and R. Rolando, Solving a Real World Project Scheduling Problem with a Genetic Approach, Belgian Journal of Operations Research, 65-78, Statistics and Computer Science, 33(1-2), 1994 .

[D59] E. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1, 269-271, 1959.

[D50] R.P. Dilworth, A decomposition theorem for partially ordered sets, Ann. Math. 51 (1950), 161-166.

[F99] A. Fehnker, Scheduling a Steel Plant with Timed Automata, *Proc. RTCSA'99*, 1999.

[Fr82] S. French, Sequencing and Scheduling - An Introduction to the Mathematics of the Job-Shop, Ellis Horwood, John-Wiley & Sons, New York, 1982.

[F73a] M.L. Fisher, Optimal Solution of Scheduling Problems using Lagrange Multipliers: Part I, 1114-1127, Operations Research, vol 21, 1973.

[F73b] M.L. Fisher, Optimal Solution of Scheduling Problems using Lagrange Multipliers: Part II, Symposium on the Theory of Scheduling and its Applications, Springer, Berlin, 1973.

[FT63] H. Fisher, and G.L Thompson, Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules, in J.F. Muth, and G.L Thompson, 225-251, (eds) Industrial Scheduling, Prentice Hall, Englewood Cliffs, New Jersey, Ch 15, 1963.

[GJ79] M. R. Garey and D. S Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[GT60] B. Giffler, and G.L. Thompson, Algorithms for Solving Production Scheduling Problems, 487-503, Operations Research, 8(4), 1960.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, Symbolic Model-checking for Real-time Systems, *Information and Computation* 111, 193–244, 1994.

[HLP93] D.J. Hoitomt, P.B. Luh, and K.R. Pattipati, Practical Approach to Job-Shop Scheduling Problems, 1-13, IEEE Trans Rob Autom, Feb, 9(1), 1993.

[JM99] A.S. Jain and S. Meeran, Deterministic Job-Shop Scheduling: Past, Present and Future, *European Journal of Operational Research* 113, 390-434, 1999.

[J55] J. R. Jackson, Scheduling a Production Line to Minimize Maximum Tardiness, Research Report 43, *Management Sciences Research Project*, UCLA, 1955.

[KPSY99] Y. Kesten, A. Pnueli, J. Sifakis and S. Yovine, Decidable Integration Graphs, *Information and Computation* 150, 209–243, 1999.

[KSSW95] K. Krüger, N.V. Shakhlevich, Y.N. Sotskov, and F. Werner, A Heuristic Decomposition Algorithm for Scheduling Problems on Mixed Graphs, 1481-1497, Journal of the Operational Research Society, vol 46, 1995.

[M99] O. Maler, On the Problem of Task Scheduling, Draft, February 1999.

[Ma96] P.D. Martin, A Time-Oriented Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem, Ph. D. Thesis, School of Operations Research & Industrial Engineering, Cornell University, Ithaca, New York 14853-3801, August 1996.

[MPS95] O. Maler, A. Pnueli and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems, *Proc. STACS'95,* 229-242, LNCS 900, Springer, 1995.

[MV94] J. McManis and P. Varaiya, Suspension Automata: A Decidable Class of Hybrid Automata, in D.L Dill (Ed.), *Proc. CAV'94*, 105-117, LNCS 818, Springer, 1994.

[M60] A.S. Manne, On the Job-Shop Scheduling Problem, 219-223, Operations Research, vol 8, 1960.

[NTY00] P. Niebert, S. Tripakis S. Yovine, Minimum-Time Reachability for Timed Automata, *IEEE Mediteranean Control Conference*, 2000.

[NY00] P. Niebert and S. Yovine, Computing Optimal Operation Schemes for Chemical Plants in Multi-batch Mode, *Proc. HSCC'2000*, 338-351, LNCS 1790, Springer, 2000.

[NP98] W.P.M. Nuijten, and C. Le Pape, Constraint-Based Job Shop Scheduling with ILOG SCHEDULER, Journal of Heuristics, March, 3(4), 271-286, 1998.

[NA96] W.P.M. Nuijten, and E.H.L. Aarts, A Computational Study of Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling, European Journal of Operational Research, vol 90, 269-284, 1996.

[PB96] C. Le Pape and P. Baptiste, A Constraint-Based Branch-and-Bound Algorithm for Preemptive Job-Shop Scheduling, *Proc. of Int. Workshop on Production Planning and Control*, Mons, Belgium, 1996.

[PB97] C. Le Pape and P. Baptiste, An Experimental Comparaison of Constraint-Based Algorithms for the Preemptive Job-shop Sheduling Problem, *CP97 Workshop on Industrial Constraint-Directed Scheduling*, 1997.

[PT96] E. Pesch, U.A.W. Tetzlaff, Constraint Propagation Based Scheduling of Job Shops, INFORMS Journal on Computing, Spring, 8(2), 144-157, 1996.

[PC95] M. Perregaard, and J. Clausen, Parallel Branch-and-Bound Methods for the Job-Shop Scheduling Problem, Working Paper, University of Copenhagen, Copenhagen, Denmark, 1995.

[RS64] B. Roy, and B. Sussmann, Les Problèmes dOrdonnancement avec Contraintes Disjonctives, Note D.S. no. 9 bis, SEMA, Paris, France, Décembre, 1964.

[SB97] I. Sabuncuoglu, and M. Bayiz, A Beam Search Based Algorithm for the Job Shop Scheduling Problem, Research Report: IEOR-9705, Department of Industrial Engineering, Faculty of Engineering, Bilkent University, 06533 Ankara, Turkey, European Journal of Operational Research, 1997.

[V91] S. Van De Velde, Machine Scheduling and Lagrangian Relaxation, Ph. D. Thesis, CWI Amsterdam, The Netherlands, 1991.

[WH92] H. Wong-Toi and G. Hoffmann, The Control of Dense Real-Time Discrete Event Systems, Technical report STAN-CS-92-1411, Stanford University, 1992.

[YI99] K. Yu-Kwong, A. Ishfaq, Benchmarking and Comparison of the Task Graph Scheduling Algorithms. Journal of Parallel and Distributed Computing 59(3), 381-422 , 1999.

[Y97] S. Yovine, Kronos: A Verification Tool for Real-time Systems, *Int. J. of Software Tools for Technology Transfer* 1, 1997.