Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 354 (2006) 272–300
Theoretical Computer Science
www.elsevier.com/locate/tcs

Scheduling with timed automata☆

Yasmina Abdeddaïm[a], Eugene Asarin[b], Oded Maler[c,*]

[a] *ESIEE, Cité Descartes, 2 Bd Blaise-Pascal, 93162 Noisy-le-Grand, France*
[b] *LIAFA, Université Paris 7, 2 place Jussieu, 75251, Paris, France*
[c] *VERIMAG, 2 avenue de Vignate, 38610, Gières, France*

Abstract

In this work, we present timed automata as a natural tool for posing and solving scheduling problems. We show how efficient shortest path algorithms for timed automata can find optimal schedules for the classical job-shop problem. We then extend these results to synthesize adaptive scheduling strategies for problems with uncertainty in task durations.

Keywords: Scheduling; Verification and synthesis; Timed automata

1. Introduction

At the most abstract level the problem of scheduling can be defined as follows. A set P of tasks is to be performed using a bounded set M of available and reusable resources. Each task is characterized by its duration, by the resources it needs in order to execute and by precedence relationships it has with other tasks. A conflict between two or more tasks occurs when their simultaneous demand for some type of resource exceeds the availability of that resource. A scheduler has to resolve such conflicts by deciding to which of the competing tasks to give the resource first and which tasks will have to wait until the resource is released. Different schedules lead naturally to different orders of task execution and the goal of optimal scheduling is to find a scheduler such that the behavior it induces is the best according to some evaluation criterion.

Variations on this problem appear in almost any application domain. The original motivating application comes from industrial engineering: how to use a finite number of machines in a factory in order to manufacture different products efficiently. A smaller scale variation on this problem is the preparation of a meal consisting of several courses, each has to be prepared according to a recipe while using a finite number of heat sources, containers and tools. Scheduling of trains (respectively, airplanes) is done by allocating tracks and junctions (respectively, air corridors and landing tracks) to different trains or airplanes at different times. Other instances of scheduling occur while assigning human resources to different tasks in a project. In computer science and engineering alone, scheduling problems occur at

☆ This work was partially supported by European Community Esprit-LTR project 26270 VHS (Verification of Hybrid systems), the AFIRST French-Israeli collaboration project 970MAEFUT5 (Hybrid Models of Industrial Plants) and the European Community projects IST-2001-35304 AMETIST (Advanced Methods for Timed Systems), and IST-2001-33520 CC (Control and Computation).
* Corresponding author.
E-mail addresses: abdedday@esiee.fr (Y. Abdeddaı¨m), asarin@liafa.jussieu.fr (E. Asarin), maler@imag.fr (O. Maler).
URLs: http://www.liafa.jussieu.fr/∼asarin (E. Asarin), http://www.verimag.imag.fr/∼maler (O. Maler).

0304-3975/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2005.11.018

various levels such as the allocation of CPU time and peripheral devices in a multi-tasking operating systems, the allocation of registers in a CPU or the allocation of communication channels in a network.

The diversity of scheduling problems and the fact that they are treated by different scientific and engineering communities led to the undesired situation where similar problems are solved using domain-specific and often ad hoc methods and where solutions are re-invented each time without leading to the emergence of a unified scheduling theory. Perhaps the only discipline which came close to building an application-independent, mathematical and algorithmic theory of scheduling is Operation Research where scheduling is formulated as a certain type of a combinatorial optimization problem. However, as we argue in this paper, this approach is not the most natural one for expressing some complex scheduling situations that occur in real life.

The work reported in this paper is a first step in the development of an alternative general theory of scheduling inspired by the methodology of verification and based on the timed automaton model. We feel that much of the success in verification is due to its use of *state-space* based *dynamic* models, such as automata, to represent the systems to be analyzed (digital circuits and finite-state programs). The principles of verification as we see them can be summarized as follows:

(1) Each component of the system in question is modeled as an automaton where the next state is determined as a function of the current state and possible interactions with states and events of other components.
(2) In these models it is possible to make distinctions between *controlled* and *uncontrolled* actions, that is, those initiated by the component and those coming from its outside environment (in some contexts these are also called *disturbances*).
(3) The semantics of the system is defined by the set of all behaviors that it can generate, namely sequences of states and events that follow the dynamics of each component and satisfy their interaction constraints.
(4) Each behavior can be evaluated according to whether it satisfies some desired properties expressed in some formalism for describing sets of sequences.
(5) The whole system is evaluated according to the evaluation of some/all of its behaviors.
(6) The evaluation can be done in a variety of ways ranging from algorithmic verification which practically computes all possible behaviors, to deductive verification which attempts to give "analytic" proofs of some claims about these behaviors.

In contrast, many approaches to timing related problems, such as those used in Operation Research, AI or Queuing Theory, pay less attention to the explicit modeling of the system dynamics but rather reduce the scheduling problem into some type of optimization or constraint satisfaction problem. The choice of problem formulation is, more often then not, driven by the existence of certain known results and algorithms, rather than by the faithfulness of the model to the phenomenon under study. [1] Such methods can be extremely successful in solving particular problems efficiently, but their rigid nature can prevent their reusability. We strongly believe that if scheduling is to become a more mature discipline, its approach to problem solving should be based on modeling problems faithfully by a clean semantic model, and not in terms of the specific technique used to solve them. Such an approach does not, of course, change the inherent computational complexity of the problem, but it provides more freedom in choosing the solution method that gives the best trade-off between its computational complexity and the quality of the solution it provides.

Another advantage of the automaton-based approach is that it enables the user to formulate, in a very natural fashion, distributed systems comprising of small interacting sub-systems. In other approaches one does not have such an intuitive notion of communicating sub-systems but rather a very large number of equations and inequalities in which the dynamical and compositional aspects are less explicit. Yet another advantage of this dynamic state-space approach is that it provides an "executable" operational model that interacts well with the actual evolution of the schedule and hence allows better execution monitoring, interaction with human operators and adaptiveness in general.

In order to adapt the conceptual and algorithmic tools of verification methodology we need to extend it in several directions. First we have to use a dynamic model that can express effectively the *quantitative timing information* associated with the *duration* of tasks (other non-temporal quantitative aspects of scheduling are outside the scope of this

[1] Of course, the phenomenon of "when you have a hammer everything looks like a nail" is not particular to scheduling and optimization and the present authors might be suffering from it as well—hopefully with a more generic hammer.

paper). For this purpose we use the timed automaton, an extension of the automaton operating on the real-time domain, which has established itself in the last decade as the object of choice for modeling and analyzing time-dependent phenomena. [2] The clocks in the timed automaton encode into the state exactly the information necessary to determine the future: each clock represents the time that has elapsed since the occurrence of a certain past event (beginning of the execution of a task) upon which a future event (the termination of the task) depends.

Secondly, we have to extend the way behaviors are typically evaluated in verification (correct vs. incorrect) to cover *quantitative* measures such as the time or cost associated with each behavior. Finally, we have to adapt verification algorithms which, typically, take the system as *given*, to become *synthesis* algorithms, that is, not using them to evaluate a given schedule but to synthesize an optimal schedule from a model that includes all possible schedules.

In this paper, we start with the classical job-shop scheduling problem studied extensively during the last decades. This problem is very simple to formulate yet it exhibits the inherent complexity of scheduling as a problem where the exponentially growing number of discrete choices dominate the rather simple linear algebra involved. Our first exercise is to show that this problem can be reduced to the problem of finding shortest paths in timed automata. While developing the algorithm we have discovered the concept of *non-lazy schedules* which allows us to restrict our attention to a finite subset of the non-countable set of possible schedules. The implementation of the algorithm and of various related heuristics demonstrates experimentally that no severe performance penalty is associated with the automaton-based approach.

In the second part of the paper we demonstrate the conceptual merits of our approach by posing and solving an extension of the job-shop problem in which task durations admit a bounded uncertainty. After defining the appropriate criterion of optimality, we develop an algorithm in the dynamic programming style which finds adaptive scheduling strategies that are optimal in this sense. We believe that these examples will convince the reader in the viability of our approach.

The rest of the paper is organized as follows. In Section 2 we introduce the job-shop scheduling problem. Section 3 presents timed automata and shows how they model scheduling problems in a most natural way. Section 4 is devoted to the algorithmics of finding shortest paths in timed automata including the underlying result concerning non-lazy schedules, improved search methods and experimental results. In the second part we move to scheduling under uncertainty. In Section 5 we discuss the problem of evaluating the performance of an open systems, describe the problem of scheduling under bounded temporal uncertainty, show why simple worst-case reasoning is not interesting for this problem and define the appropriate optimality criterion. In Section 6 this problem is formulated and solved algorithmically using timed automata and some experimental results are reported. In Section 7, we sketch a solution of a probabilistic variant of the problem with exponentially distributed task durations. Finally, we survey some related work and suggest further research directions.

This paper is based on the Ph.D. thesis [1] and the conference papers [4,2].

2. Deterministic 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 it is a good representative 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 in [29], remained unsolved for almost 25 years, in spite of the research effort spent on it).

A job-shop problem consists of a finite set $J = \{J^1, \ldots, J^n\}$ of jobs to be processed on a finite set $M = \{m_1, \ldots, m_k\}$ of machines. Each job J^i is a finite sequence of tasks to be executed one after the other, where each task is characterized by a pair of the form (m, d) with $m \in M$ and $d \in \mathbb{N}$, indicating the required utilization of machine m for a fixed time

[2] It should be noted that timed automata are not the only possible dynamic model for timing related behaviors. In principle, this work could as well be phrased in terms of some variant of timed Petri nets (see a survey of those in [16]). It is a matter of taste whether one prefers to view interaction as communication between automata or via token passing. Whatever the formalism chosen, the final object to be analyzed is the same regardless of whether it is a product of automata or a marking graph of a Petri net. The theoretic and algorithmic results of the current paper (as well as many other results in verification) could have been derived, in principle, from a PN formulation, but in reality they have not. Whether this is due to inherent properties of the models or of the communities remains an open question.

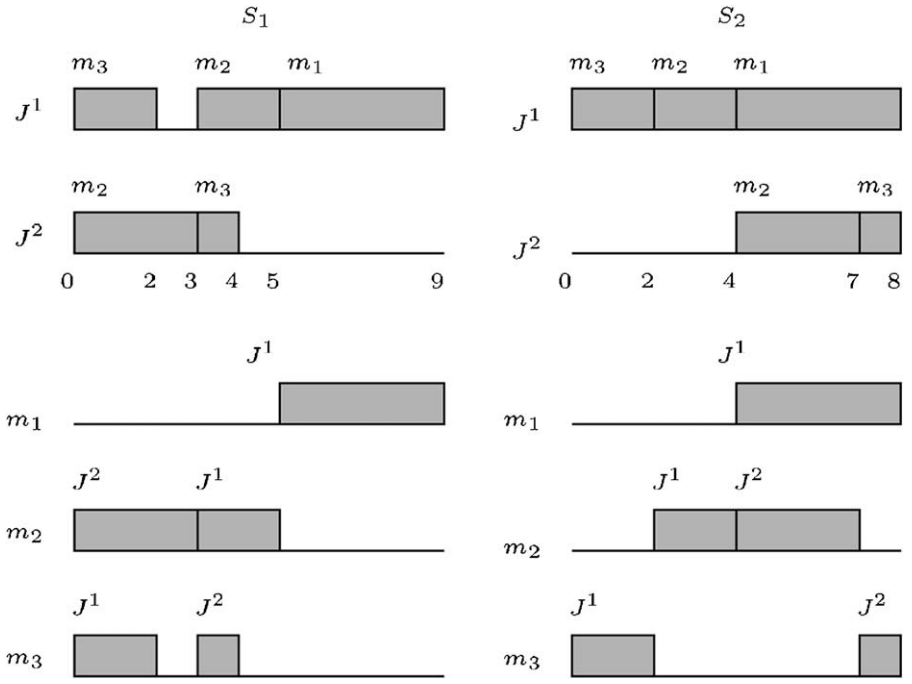

Fig. 1. Two feasible schedules S_1 and S_2 visualized as the task progress (up) and as machine allocation (down).

duration d. Each machine can process at most one task at a time and, due to precedence constraints, at most one task of each job may be processed at any time. Tasks cannot be preempted once started.

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

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

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

Two schedules S_1 and S_2 are depicted in Fig. 1. Schedules are three-dimensional objects involving tasks, machines and time and hence they can be depicted using two types of Gantt diagrams based either on job progress or on machine occupation. The first form is more related to automaton modeling of the problem and will be used henceforth. The length of S_2 is $|S_2| = 8$ and it is the optimal schedule.

Note that a job can be idle at time t even if its precedence constraints are satisfied and the machine it needs at that time is available. As one can see in schedule S_2 of Fig. 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 tasks 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 possible solutions that need to be explored and is the major source of the complexity of scheduling.

Our problem definition below is slightly more general than the classical job-shop problem, allowing precedence constraints that are not necessarily a set of linear chains.

Definition 1. (Machine scheduling problem). A machine scheduling problem $\mathcal{J} = (P, \prec, M, \mu, d)$ consists of a set $P = \{p_1, \ldots, p_m\}$ of tasks, a strict partial-order precedence relation $\prec$ on P, a set $M = \{m_1, \ldots, m_n\}$ of machines, a function $\mu : P \to M$ assigning machines to tasks and a duration function $d : P \to \mathbb{N}$.

We assume throughout the paper that all machines are distinct and that each task p can be performed only on machine $\mu(p)$. The extension of the model to the case where a task can be executed on one out of several machines (possibly with different speeds) is an easy exercise that can be done at the expense of complicating the notation.

We denote by $\Pi(p)$ the set of immediate predecessors of p, i.e. those p' such that $p' \prec p$ and there is no p'' such that $p' \prec p'' \prec p$.

We want to find the schedule that minimizes the total execution time and respects the following conditions: (1) a task can be executed only if all its predecessors have terminated; (2) each machine can process at most one task at a time; (3) tasks cannot be preempted once started.

Definition 2. (Feasible and optimal schedules). A schedule for a problem $\mathcal{J} = (P, \prec, M, \mu, d)$ is determined by the function $st : P \to \mathbb{R}_+$ indicating the start time of each task. In a deterministic setting, the end time of a task is $en(p) = st(p) + d(p)$. A schedule is feasible if it satisfies:

(1) Precedence: For every $p, p' \in P$, $p \prec p' \Rightarrow en(p) \leqslant st(p')$.
(2) Mutual exclusion: For every two tasks p, p' such that $\mu(p) = \mu(p')$, $[st(p), en(p)] \cap [st(p'), en(p')] = \emptyset$.

The length of the schedule is $\max\{en(p) : p \in P\}$. An optimal schedule is a schedule whose length is minimal.

Note that condition (2) reduces into a *disjunction*

$$st(p) - st(p') \geqslant d(p') \vee st(p') - st(p) \geqslant d(p)$$

rendering the whole problem highly non-convex when viewed as a constrained optimization problem.

3. Modeling with timed automata

Timed automata [9] are automata augmented with continuous clock variables whose values grow uniformly at every state. Clocks can be reset to zero at certain transitions and tests on their values can be used as conditions for enabling transitions. Hence they are ideal for describing concurrent time-dependent behaviors. Our definition below is an "open" version of timed automata which can refer to the states of other automata, ranging over Q', in their transition guards. The clocks constraints that we use are slightly less general than in the standard definition of timed automata.

Definition 3. (Timed automaton). An open timed automaton is $\mathcal{A} = (Q, C, I, \Delta, s, f)$ where

- Q is a finite set of states;
- C is a finite set of clocks;
- I is the staying condition (invariant), assigning to every $q \in Q$ a conjunction I_q of inequalities of the form $c \leqslant u$, for some clock c and integer u;
- Δ is a transition relation consisting of elements of the form (q, ϕ, ρ, q') where
  - q and q' are states;
  - $\phi = \phi_1 \wedge \phi_2$ is the transition guard where ϕ_1 is a formula characterizing a subset of an external set of states Q' and ϕ_2 is a conjunction of constraints of the form $(c \geqslant l)$ for some clock c and some integer l;
  - $\rho \subseteq C$ is a set of clocks to be reset;
- s and f are the initial and final states, respectively.

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 such valuations by V and $\mathbf{v}(c_i)$ by v_i. A configuration of the automaton is hence a pair $(q, \mathbf{v})$ consisting of a discrete state (also known as *location*) and a clock valuation. Every subset $\rho \subseteq C$ induces a reset function $Reset_\rho$ defined for every clock valuation $\mathbf{v}$ and every clock variable $c \in C$ as

$$Reset_\rho\, \mathbf{v}(c) = \begin{cases} 0 & \text{if } c \in \rho, \\ \mathbf{v}(c) & \text{if } c \notin \rho. \end{cases}$$

That is, $Reset_\rho$ 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.

A *step* of the automaton is one of the following:

- A discrete step: $(q, \mathbf{v}) \xrightarrow{0} (q', \mathbf{v}')$, where there exists $\delta = (q, \phi_1 \wedge \phi_2, \rho, q') \in \Delta$, such that the external environment satisfies ϕ_1, $\mathbf{v}$ satisfies ϕ_2 and $\mathbf{v}' = \mathit{Reset}_\rho(\mathbf{v})$.
- A time step: $(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1})$, $t \in \mathbb{R}_+$ and $\mathbf{v} + t\mathbf{1}$ satisfies I_q.

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 $t_1 + t_2 + \cdots + t_n$. Note that discrete transitions take no time.

Our goal is to model each scheduling problem using a timed automaton so that every run corresponds to a feasible schedule and the shortest run gives the optimal schedule. As a running example consider the problem $M = \{m_1, m_2\}$, $P = \{p_1, p_2, p_3\}$, $p_1 \prec p_2$, $\mu(p_1) = \mu(p_3) = m_1$, $\mu(p_2) = m_2$, $d(p_1) = 4$, $d(p_2) = 5$ and $d(p_3) = 3$.

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 to zero and can be taken only if all the automata corresponding to the tasks in $\Pi(p)$ are in their final states. The transition from p to $\underline{p}$ is taken when $c = d(p)$.

Definition 4. (Timed automaton for a task). For every task $p \in P$ its associated timed automaton is $\mathcal{A} = (Q, \{c\}, I, \Delta, s, f)$ with $Q = \{p, \overline{p}, \underline{p}\}$ where the initial state is $\overline{p}$ and the final state is $\underline{p}$. The staying conditions are *true* for $\overline{p}$ and $\underline{p}$ and $c \leqslant d(p)$ in p. The transition relation Δ consists of the two transitions:

$$\mathit{start}: \quad \left(\overline{p}, \bigwedge_{p' \in \Pi(p)} \underline{p'}, \{c\}, p\right)$$

and

$$\mathit{end}: \quad (p, c = d(p), \emptyset, \underline{p}).$$

Note that the clock is *active* only in state p where it measures the time elapsed since it started executing while its value in $\overline{p}$ does not influence the future and hence need not be part of the system state. This fact can also be deduced from observing that the only transition outgoing from $\overline{p}$ resets the clock to zero without testing its value.[3] The automata $\mathcal{A}^{p_1}$, $\mathcal{A}^{p_2}$ and $\mathcal{A}^{p_3}$ corresponding to the tasks in the example appear in Fig. 2.

To obtain the timed automaton representing the whole scheduling problem we need to compose the automata for the individual tasks. The composition takes care of the precedence constraints by allowing the automaton to make a start transition only when the automata for its predecessors are in their respective final states. Mutual exclusion constraints are enforced by forbidding global states in which two or more tasks that use the same machine are active. An n-tuple $q = (q^1, \ldots, q^n)$ is said to be *conflicting* if it contains two components q^j and q^k such that $q^j = p_j$, $q^k = p_k$ and $\mu(p_j) = \mu(p_k)$.

Definition 5. (Mutual exclusion composition). Let $\mathcal{J} = (P, \prec, M, \mu, d)$ be a machine scheduling problem and let $\mathcal{A}^i = (Q^i, C^i, I^i, \Delta^i, s^i, f^i)$ be the automaton corresponding to each task p_i. Their mutual exclusion composition is the automaton $\mathcal{A} = (Q, C, I, \Delta, s, f)$ such that Q is the restriction of $Q^1 \times \cdots \times Q^n$ to non-conflicting states, $C = C^1 \cup \cdots \cup C^n$, $s = (s^1, \ldots, s^n)$, $f = (f^1, \ldots, f^n)$, the staying condition for a global state $q = (q^1, \ldots, q^n)$ is $I_q = I_{q^1} \wedge \cdots \wedge I_{q^n}$ and the transition relation Δ contains all the tuples of the form

$$((q^1, \ldots, q^j, \ldots, q^n), \phi_2, \rho, (q^1, \ldots, r^j, \ldots, q^n)),$$

such that the source and target states are non-conflicting, $(q^j, \phi_1 \wedge \phi_2, \rho, r^j) \in \Delta^j$ for some j and ϕ_1 is satisfied by $(q^1, \ldots, q^n)$.

[3] Clock activity analysis was introduced in [25] to reduce the dimensionality of the clock space.

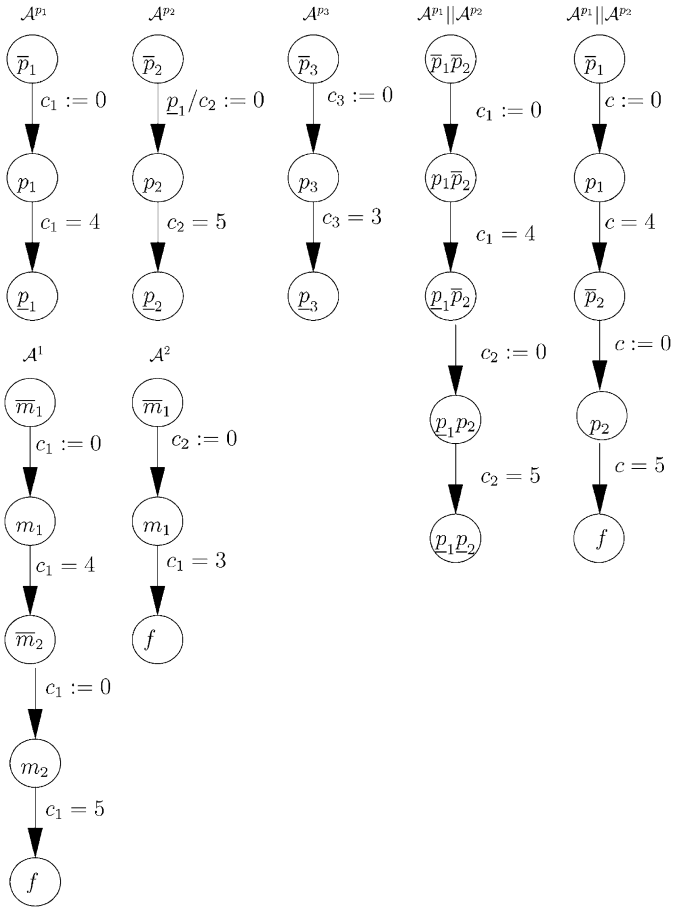

Fig. 2. Automata for tasks and jobs.

In the automata derived from tasks, the formula ϕ_1 in the guard for the *start* transition specifies that the automata for the preceding tasks are in their respective final states and the runs of the product automaton satisfy precedence constraints by construction. A run of $\mathcal{A}$ is *complete* if it starts at $(s, \mathbf{0})$ and the last step is a transition to f. From every complete run ξ one can derive in an obvious way a schedule where $st(p_i)$ is the time the $start_i$ transition is taken. The length of the schedule coincides with the metric length of ξ. Note that the interleaving semantics inserts some redundancy as there could be more than one run associated with a feasible schedule in which several tasks start or end simultaneously.

Before showing the product let us discuss the simplifications associated with the fact that we work with the job-shop problem which constitutes a special case of machine scheduling where P can be partitioned into a set $J = \{J^1, \ldots, J^n\}$ of chains called jobs, each of the form $p_1 \prec \cdots \prec p_k$. In this case each task has at most one immediate predecessor denoted by $\pi(p)$. If we look at the composition of $\mathcal{A}^{p_1}$ and $\mathcal{A}^{p_2}$ (the automaton $\mathcal{A}^{p_1}||\mathcal{A}^{p_2}$ of Fig. 2) we see that it has a chain structure because p_2 cannot move until p_1 terminates and the automaton is isomorphic to the automaton $\mathcal{A}^{p_1||p_2}$ where the states are associated with the waiting and active states of each task plus a special state f indicating the termination of the last task in the chain. For the same reason one clock is sufficient for each chain. In the rest of the paper we will draw the automata for the jobs as in $\mathcal{A}^1$ and $\mathcal{A}^2$ of Fig. 2, replacing p by $\mu(p)$. Likewise, we will specify jobs as sequences like

$$(\mu(p_1), d(p_1)), \ldots, (\mu(p_k), d(p_k)).$$

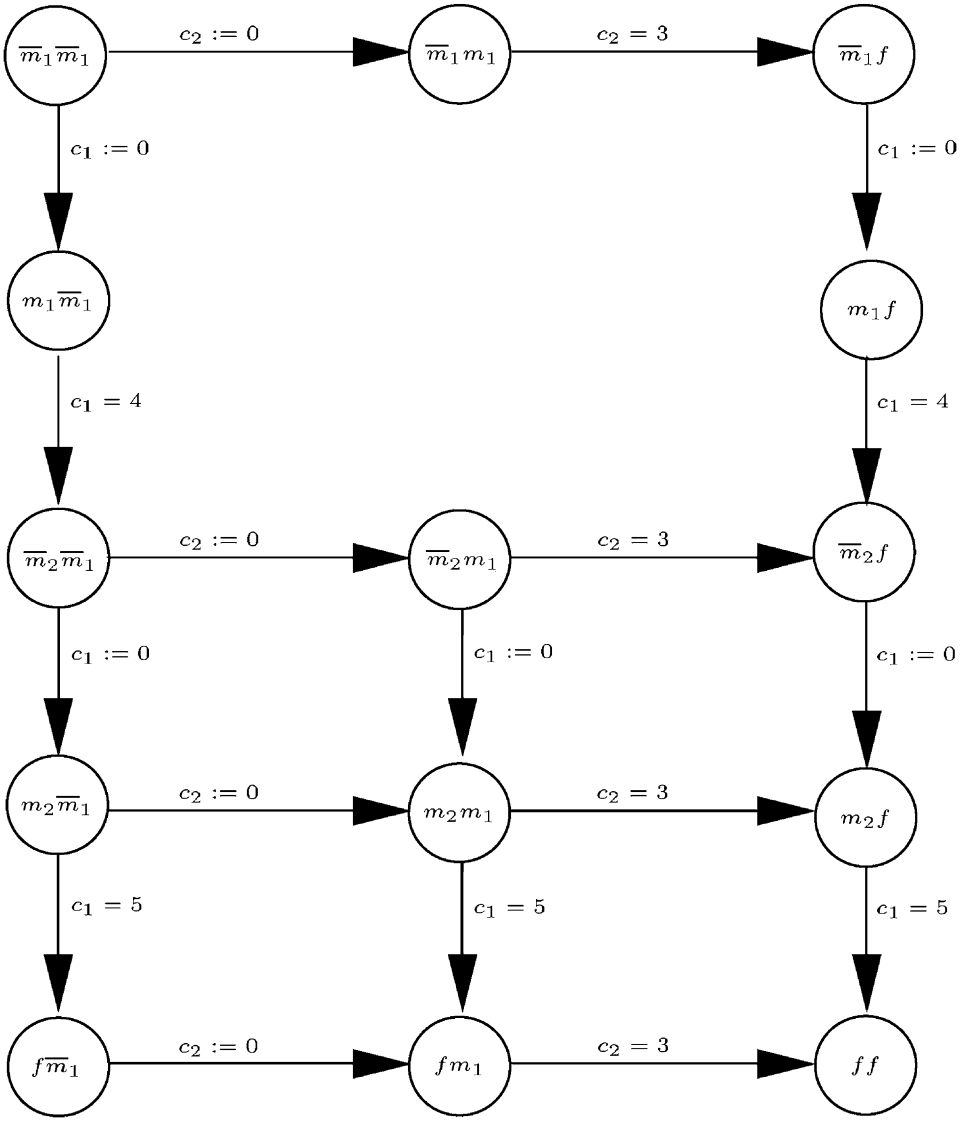

Fig. 3. The global timed automaton for the two jobs.

The global automaton obtained by composing $\mathcal{A}^1$ and $\mathcal{A}^2$ is depicted in Fig. 3. Two feasible schedules for this problem appear in Fig. 4. The length of S_1 is 9 and it is the optimal schedule for this problem. The two schedules correspond to the following two runs of the automaton (we use notation $\perp$ to indicate inactive clocks):

$$S_1 : (\overline{m}_1, \overline{m}_1, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_1, 0, \perp) \xrightarrow{4} (m_1, \overline{m}_1, 4, \perp) \xrightarrow{0} (\overline{m}_2, \overline{m}_1, \perp, \perp) \xrightarrow{0}$$
$$(m_2, \overline{m}_1, 0, \perp) \xrightarrow{0} (m_2, m_1, 0, 0) \xrightarrow{3} (m_2, m_1, 3, 3) \xrightarrow{0} (m_2, f, 3, \perp) \xrightarrow{2} (m_2, f, 5, \perp) \xrightarrow{0} (f, f, \perp, \perp),$$

$$S_2 : (\overline{m}_1, \overline{m}_1, \perp, \perp) \xrightarrow{0} (\overline{m}_1, m_1, \perp, 0) \xrightarrow{3} (\overline{m}_1, m_1, \perp, 3) \xrightarrow{0} (\overline{m}_1, f, \perp, \perp) \xrightarrow{0}$$
$$(m_1, f, 0, \perp) \xrightarrow{4} (m_1, f, 4, \perp) \xrightarrow{0} (\overline{m}_2, f, \perp, \perp) \xrightarrow{0} (m_2, f, 0, \perp) \xrightarrow{5} (m_2, f, 5, \perp) \xrightarrow{0} (f, f, \perp, \perp).$$

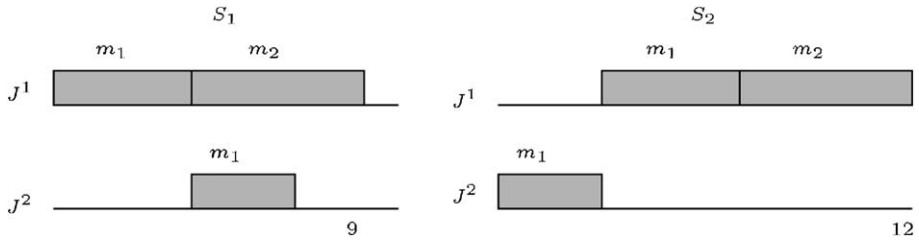

Fig. 4. Two schedule S_1 and S_2 for the example.

4. Shortest paths in timed automata

The standard forward reachability algorithm for timed automata [33], used in tools such as Kronos, Uppaal and IF [46,36,20], can compute the set of all reachable configurations of a given automaton. In order to compute the shortest path one can augment an automaton $\mathcal{A}$ with an additional clock T which is never reset to zero and hence it measures the time elapsed since the beginning of a run.[4] Clearly, a configuration $(q, \mathbf{v})$ is reachable within time t in $\mathcal{A}$ iff $(q, \mathbf{v}, t)$ is reachable in the augmented automaton. Hence, reachability computation is sufficient for solving the shortest path problem (see also [8]). This solution is, however, not very efficient for the following reason. The reachability algorithm was designed with *verification* in mind and, consequently, it is *exhaustive* in the sense that it computes all possible runs of the automaton. These runs cover all (qualitative) paths in the automaton, and in each path they cover all the uncountably-many choices of times in which a transition could be taken. Thus the algorithm has to manipulate an exponential number of zones (special polyhedra in the clock space represented by a data-structure of size quadratic in the number of jobs). As we will see, in our case, a much more efficient algorithm is possible.

We start with an observation concerning optimal schedules that we use to eliminate the need for zones. A task p is *enabled* at time t in a given schedule if $t \in [t_1, t_2]$ where $t_1 = en(\pi(p))$, $t_2 = st(p)$ and the machine $\mu(p)$ is not used by any other task. We say that a schedule S exhibits *laziness* at task p if p is enabled in a non-empty interval $[t, st(p)]$. A schedule is lazy if it exhibits laziness at one or more task. We have noted before that sometimes it is preferable *not* to start a task as soon as it is enabled, however, *this waiting is useless if no other task takes advantage of it*.[5] This fundamental intuition is formalized below.

Claim 1 (*Non-lazy optimal schedules*). *Any lazy schedule S can be transformed into a non-lazy schedule $\hat{S}$ with $|\hat{S}| \leqslant |S|$. Hence every machine scheduling problem admits an optimal non-lazy schedule.*

Proof. The proof is by taking a lazy schedule S and transforming it into a schedule S' in which laziness occurs "later". A schedule induces a partial order relation $\sqsubset$ on P defined as $p \sqsubset p'$ if either $p \prec p'$ (when they belong to the same job) or $\mu(p) = \mu(p')$ and $st(p) < st(p')$ (when they are in conflict and the schedule gives priority to p). The laziness elimination procedure picks a lazy task p which is minimal with respect to $\sqsubset$ and shifts its start time backwards to the beginning of the laziness interval to yield a new feasible schedule S', such that $|S'| \leqslant |S|$. Moreover, the partial order associated with S' is identical to the one induced by S. The laziness at p is thus eliminated, and this might create new manifestations of laziness at later tasks which are eliminated in the subsequent stages of the procedure (see illustration in Fig. 5). Let $L(S) = \{p : \exists p' \sqsubseteq p \text{ s.t. there is laziness in } p'\}$, namely the set of tasks that are lazy or preceded by laziness. Clearly the laziness removal procedure decreases $L(S)$ and terminates due to finiteness. □

The next step is to restrict the runs of the automaton to those that correspond to non-lazy schedules. A *lazy run* in a job-shop automaton $\mathcal{A}$ is a run containing a fragment

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

[4] A similar construction was previously described in [42] to implement shortest path algorithm for cyclic timed automata using forward reachability.

[5] The situation is quite different in scheduling under uncertainty where waiting may lead to gaining additional *information*.

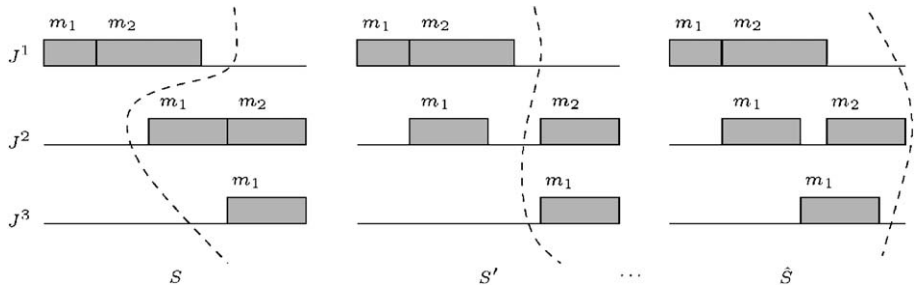

Fig. 5. Removing laziness from a schedule S: first we eliminate laziness in the task of J^2 which uses m_1. This creates further manifestation of laziness which are subsequently removed until a non-lazy schedule $\hat{S}$ is obtained. The dashed line indicates the frontier between $L(S)$ and the rest of the tasks.

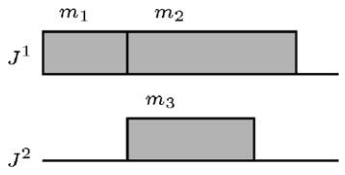

Fig. 6. A lazy schedule which corresponds to an immediate run.

such that the $start_i$ transition is enabled in all states $(q, \mathbf{v}), \ldots, (q', \mathbf{v}')$. As one can see this notion is non-local in the sense that at the moment of not taking the $start_i$ transition we do not know yet whether this run will be extended to a lazy one. To simplify the presentation we will use here the weaker notion of an *immediate* run. The actual implementation generates only non-lazy runs and the reader can find more details in [1].

Definition 6. (Immediate runs). An immediate run is a run in which whenever a *start* transition is taken in a state, it is taken as soon as it is enabled. A non-immediate run contains a fragment

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

Note that enabledness of *start* transitions does not depend on clock values. Clearly a schedule derived from a non-immediate run exhibits laziness, hence in order to find an optimal schedule it is sufficient to explore the (finite) set of immediate runs. The converse is not true: Fig. 6 shows a lazy schedule which is immediate. It is lazy because m_3 could have started at time 0, but it corresponds to an immediate run because m_3 was started after the termination of m_1, that is, in a state different from the state where it could have been started.

The restriction to immediate runs transforms the timed automaton into a discrete directed graph where nodes correspond to *single* configurations connected by a simple successor relation defined 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.

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

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

$$Succ^t(q_1, \ldots, q_n, v_1, \ldots, v_n, t) = \{(q'_1, \ldots, q'_n, v'_1, \ldots, v'_n, t + \theta)\},$$

such that for every i

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

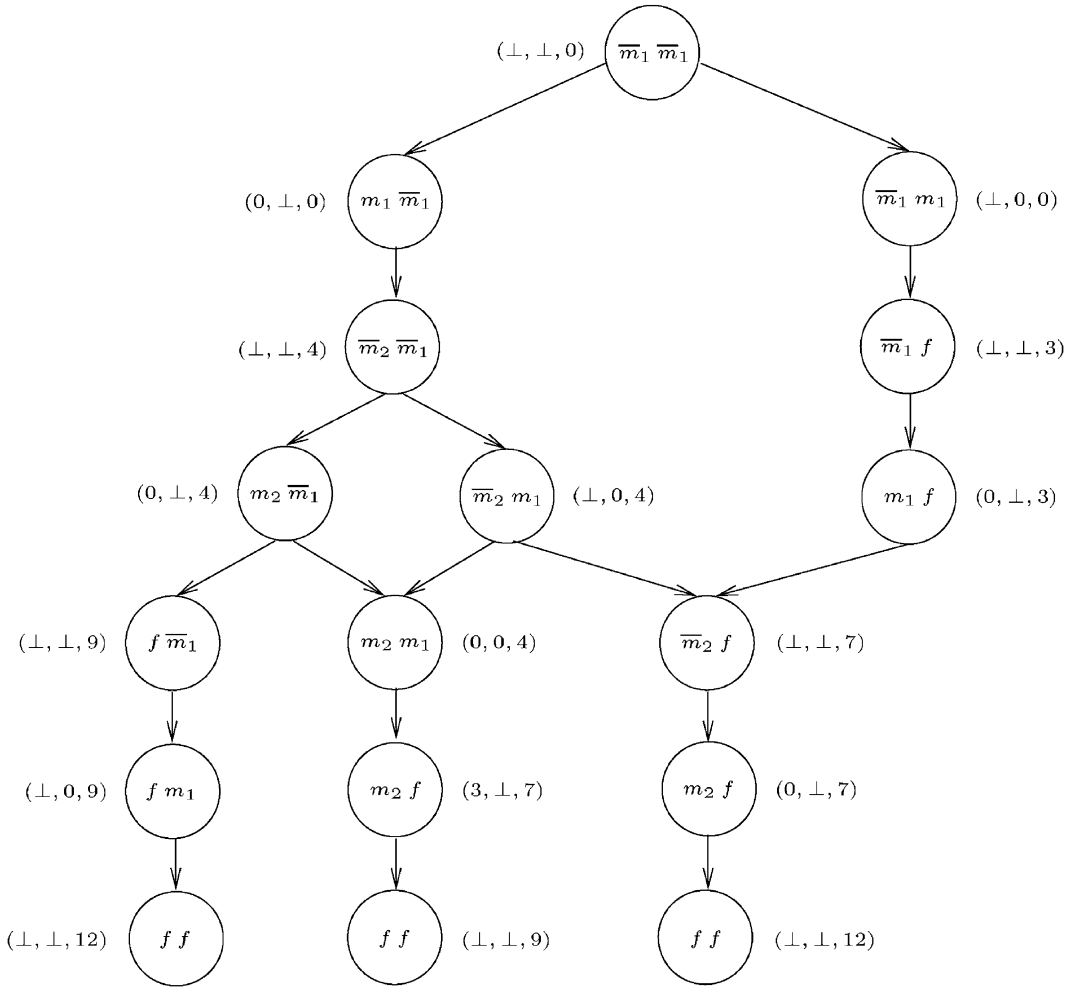

Fig. 7. The immediate runs of the timed automaton of Fig. 3.

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

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

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

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

Fig. 7 shows the graph thus obtained from the automaton of Fig. 3, where the paths correspond to the 5 immediate runs. Note that due to interleaving the same schedule can be represented by more than one run. Applying standard search algorithms to this graph we can find the shortest path (and the optimal schedule) *without using zone technology*.

Although using points instead of zones reduces significantly the computational cost, the inherent combinatorial explosion remains. In the rest of this section we describe further methods to reduce the search space, some of which preserve the optimal solutions and some provide sub-optimal ones. Similar ideas were first explored in [14]. The first self-evident idea is to avoid exploring identical nodes or nodes that are obviously worse than nodes already explored.

Definition 7. (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 $t' \leqslant t$ and $\mathbf{v} \geqslant \mathbf{v}'$.

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 through $(q, \mathbf{v}, t)$ which is not longer. Hence whenever we encounter a new node in the graph we check whether it is dominated by an explored or waiting node and in this case we discard it. If it dominates a node in the waiting list we replace it.

The next thing to do is to apply best-first search and explore the "most promising" nodes first. To this end we need an *evaluation function* over configurations. Consider a job $J = (p_1, d_1), \ldots, (p_k, d_k)$ and its corresponding automaton. For every configuration (q, v) of this automaton $g(q, v)$ is a lower-bound on the time remaining until f is reached from the configuration (q, v):

$$
\begin{aligned}
&g(f, \perp) = 0,\\
&g(\overline{p}_j, \perp) = \sum_{l=j}^{k} d(p_l),\\
&g(p_j, v) = g(\overline{p}_j, \perp) - v.
\end{aligned}
$$

The evaluation of global configurations is defined as

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

Note that $\max\{g\}$ gives the most optimistic estimation of the *remaining* time to completion, assuming that no job will have to wait due to a conflict. The best-first search algorithm below maintains the waiting list sorted according to $\mathcal{E}$. It 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 1 (*Best-first forward reachability*).

Waiting := {*Succ*$(s, \mathbf{0}, 0)$};
Best := ∞
$(q, \mathbf{v}, t)$:= *first in Waiting*;
while $\mathcal{E}(q, \mathbf{v}, t) <$ *Best*
do
  For every $(q', \mathbf{v}', t') \in$ *Succ*$(q, \mathbf{v}, t)$;
    if $q' = f$ **then**
      Best := $\min\{$*Best*$, t'\}$
    else
      Insert $(q', \mathbf{v}', t')$ *into Waiting*;
  Remove $(q, \mathbf{v}, t)$ *from Waiting*
  $(q, \mathbf{v}, t)$:= *first in Waiting*;
end

A prototype implementation of this algorithm can find optimal schedules for problems with up to 6 jobs and 6 machines in few seconds. To treat larger problems we resort to a heuristic algorithm which is not guaranteed to produce the optimal solution. The algorithm 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 $\mathcal{E}$) nodes, 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 small w is better than the one found with a larger one.

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 1 we compare our best results on these problems with the best results reported in Table 15 of the comprehensive survey [35], where the results of the 18 best-known methods were compared. As one can see our results are typically 5–10% longer than the optimum. For comparison, an algorithm which picks the best out of 3000 randomly generated runs deviates from the optimum by more than 100%.

Table 1
The results for 10 hard problems using the bounded width heuristic

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

The first three columns give the problem name, number of jobs and number of machines (and tasks). Our results (time in seconds, the length of the best schedule found and its deviation from the optimum) appear next.

5. Scheduling under uncertainty

The problem treated so far was completely *deterministic*. All the information concerning the tasks to be executed was known in advance, including their identity, inter-dependence, duration and release time. The same goes for the machines whose quantity was assumed to be fixed. Real life is not like that. New tasks can arrive in the middle of execution while others can be canceled. Task processing can take more or less time than expected, machines may break down, cost criteria may change, etc. In such situations the actual evolution of the system depends on the actions of two "players", the scheduler which decides whether or not to start a task in a given situation and the "environment", a generic name for all sources of *uncontrolled* external events such as the arrival or termination of a task.

5.1. Strategies and their evaluation

The evaluation or optimization of the performance of such an *open* reactive system which interacts with an external environment, raises some serious conceptual problems. [6] In a deterministic setting, each scheduler induces a *unique* schedule according to which it can be evaluated and compared with other candidate schedulers. For an open system S, each instance d of the environment can potentially induce a different behavior $S(d)$, and the question is how to take all these behaviors into account while evaluating and comparing schedulers. Several approaches to this problem are commonly used:

- *Worst case*: The system is evaluated according to its worst behavior.
- *Average case*: The set of all environment instances is considered as a probability space and this induces a probability over all system behaviors. The system is then evaluated according to the expected value (over all its behaviors) of the performance measure.
- *Nominal case*: The system is evaluated based on one behavior which corresponds to one "typical" instance of the environment.

Each of these approaches has its advantages and shortcomings. The worst-case approach is often used for safety-critical systems where the cost associated with bad behaviors is too high to tolerate, even if they constitute a negligible fraction of the possible behaviors. This is implicitly the approach taken in verification, where the performance measure is discrete and consists of a binary classification into "correct" and "incorrect", and this means that a system is correct only if *all* its behaviors satisfy the property in question. On the negative side, this approach might lead to an over-pessimistic allocation of resources which can be very inefficient during most of the system lifetime. [7]

[6] Readers interested in a more comprehensive discussion of these issues are invited to look at [39].

[7] A good analogy is to live all your life wearing a helmet fearing a meteorite rain, or going to the airport a day before the flight in anticipation of all conceivable traffic jams.

The probabilistic approach is more appropriate when the performance measure is more "continuous" in nature, e.g. the waiting time in a queue, and one can tolerate some performance degradation during pressure periods. The implicit assumption underlying the nominal approach is somewhat similar to the probabilistic one, namely, the nominal behavior is "close" to most of the behaviors we are likely to see during the system life-time and the performance of other behaviors varies "continuously" with the distance from the nominal one. This approach is widely (and implicitly) used in Control Theory, for example, in "step response" analysis the system is simulated with one disturbance which is, in certain cases, sufficient for its evaluation.

From a computational standpoint the evaluation of a given scheduler S is the easiest under the nominal approach because when d is fixed the system is closed and the behavior $S(d)$ can be computed by simple simulation (it is represented by a single path in the corresponding automaton). Moreover, the comparison of two candidate systems S and S' is based on the same d. In the worst-case approach when it is not known a priori which d induces the worst behavior, one has to "simulate exhaustively" and evaluate the scheduler against all instances in order to find the worst case. This is the inherent difficulty of verification compared to testing/simulation. Moreover, when we want to compare S and S' for optimality, it might be that each of them attains its worst performance on a different instance. The probabilistic approach is a priori [8] the most difficult because not only do we need to explore all behaviors but also to keep track of their probabilities in order to compute the overall evaluation of the system.

5.2. Scheduling under temporal uncertainty

In the rest of the paper we treat a non-deterministic generalization of the job-shop scheduling problem where the exact *duration* of the tasks is not given in advance but rather restricted to be bounded within an interval of the form $[l, u]$. In Section 7, we will also treat an alternative model where the duration of each task is given as a continuous random variable. Each *instance* (or *realization* in the Operation Research jargon) of the environment consists of selecting a number $d \in [l, u]$ for every task. The behavior induced by the scheduler on each instance is evaluated, as before, according to the length of the schedule.

As an example consider the job-shop problem

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

where the only resource under conflict is m_3 and the order of its utilization is the only decision the scheduler needs to take. 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 $d = (d_1, d_2) \in [2, 8] \times [2, 4]$. It is very important to note that in our example (and in "reactive" systems in general) instances reveal themselves *progressively* during execution—the value of d_2, for example, is known *only after the termination* of the second task of J^1.

Each instance defines a deterministic scheduling problem admitting one or more optimal solutions. Fig. 8(a) depicts optimal schedules for the instances $(8, 4)$, $(8, 2)$ and $(4, 4)$. In general, only a *clairvoyant* scheduler who knows the whole instance in advance can always find such an optimal schedule.

For this particular type of problem, worst-case optimization can be reduced to nominal-case because there is one specific instance, namely the one where each task terminates as late as possible, such that the performance of any scheduler on this instance will be at least as bad as on any other instance. To obtain worst-case optimality it is sufficient to 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 scheduler for our example, based on instance $(8, 4)$, is depicted in Fig. 8(b), and one can see that is rather wasteful for other instances. Intuitively we will prefer a smarter adaptive scheduler that reacts to the evolution of the system and modifies its decisions according to additional information revealed during execution. This is the essential difference between a schedule (a plan, an open-loop controller) and a scheduling *strategy* (a reactive plan, a closed-loop 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 and the strategy can be reduced to a simple assignment of start times to tasks.

[8] At least when the approach is applied naively without using additional mathematical information that can lead to analytic solutions in some special cases.

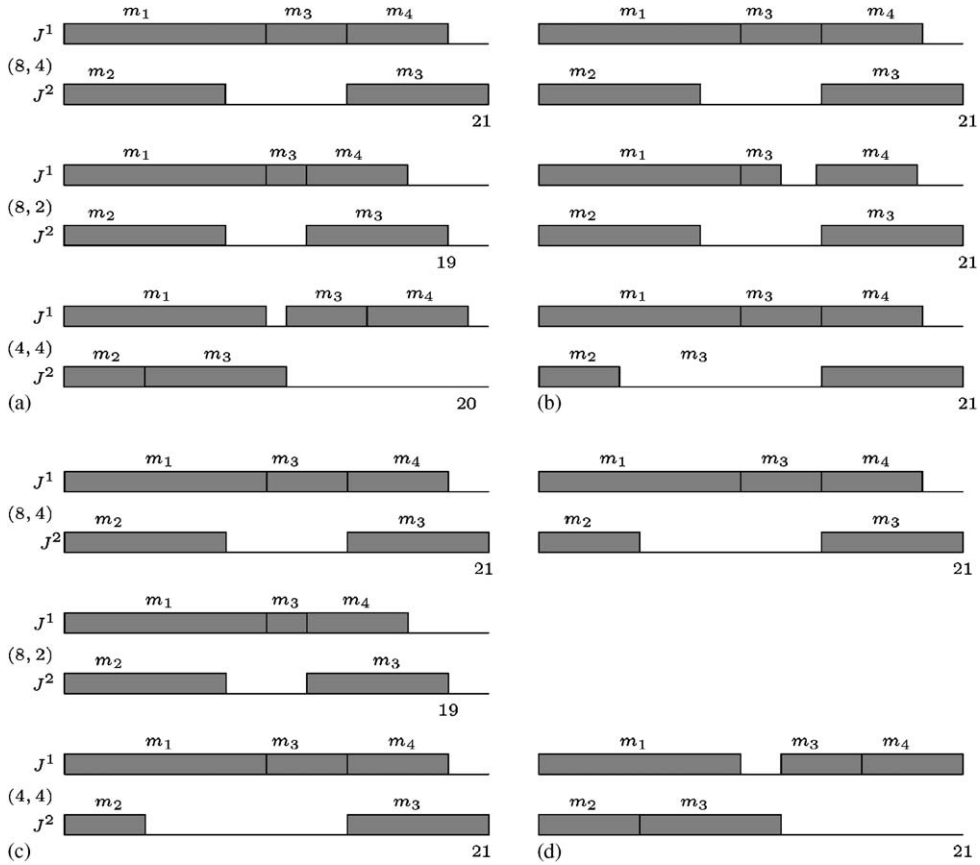

Fig. 8. (a) Optimal schedules for three instances. For the first two the optimum is obtained with $J^1 \sqsubset J^2$ on m_3 while for the third—with $J^2 \sqsubset J^1$; (b) a static schedule based on the worst instance (8, 4). It gives the same length for all instances; (c) the behavior of a hole filling strategy based on instance (8, 4); (d) the equal performance of the two strategies on instance (5, 4).

One of the simplest ways to be adaptive is the following. First we choose a *nominal instance d* and find a schedule S which is optimal for that instance. Rather than taking S "literally" as the function st, 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 $J^1 \sqsubset J^2$ on m_3. Then, during execution, we start every task as soon as its predecessors have terminated, provided that the ordering is not violated (a similar strategy was used in [43] and probably elsewhere). As Fig. 8(c) shows, such a strategy is better than the static schedule for instances such as (8, 2) where it takes advantage of 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 $J^1 \sqsubset J^2$ ordering on m_3.

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 in time as needed while maintaining the order. On the other hand, a static schedule (at least when interpreted as a function from time to actions) can only be based on the worst case—a schedule based on another nominal instance may assume a resource available at some time point, while in reality that resource 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 for 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) (Fig. 8(a)) we see that in both of them the decision whether or not to give m_3 to J^2 is taken at the same qualitative state where m_1 is executing and m_2 has terminated. The only difference is in the elapsed execution time 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 automaton models, is represented by clock values.

Consider the following approach: initially we find an optimal schedule for some nominal instance. During execution, whenever a task terminates (before or after the time it was assumed to) 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) and start executing it. If task m_2 in J^2 terminated after 4 time units we obtain the residual problem

$$J_1' = (\mathbf{m_1}, \mathbf{6}), (m_3, 4), (m_4, 5), \quad J_2' = (m_3, 7),$$

where the boldface letters indicate that m_1 must be scheduled immediately (it is already executing and we assume no preemption). For this problem the optimal solution will be to give m_3 to J^2. Likewise if m_2 terminates at 8 we have

$$J_1' = (\mathbf{m_1}, \mathbf{2}), (m_3, 4), (m_4, 5), \quad J_2' = (m_3, 7)$$

and the optimal schedule consists of waiting for the termination of m_1 and then giving m_3 to 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*. [9]

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 is based on expressing the scheduling problem using timed automata and synthesizing a controller *off-line*. In this framework [13,11,6] 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 retrieved from the strategy look-up table and the results are identical to those obtained via online re-scheduling. [10] The major contribution of this paper is the formalization of this intuition and the development and implementation of an algorithm for finding adaptive schedulers that are optimal in this sense.

5.3. Problem statement

Definition 8. (Uncertain machine scheduling). An uncertain machine scheduling problem is $\mathcal{J} = (P, \prec, M, \mu, D, U)$ where P, $\prec$, M and μ are as in Definition 1, $D : P \to Int(\mathbb{N})$ assigns an integer-bounded interval to each task and $U \subseteq P$ 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.

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: $d \leqslant d'$ if $d(p) \leqslant d'(p)$ for every $p \in P$. Any environment instance induces naturally a deterministic instance of $\mathcal{J}$, denoted by $\mathcal{J}(d)$. The worst case is defined by the maximal instance $\hat{d}$ where $\hat{d}(p) = D^u(p)$ for every p.

A *feasible schedule* for an instance $\mathcal{J}(d)$ of the problem is characterized, as in Definition 2, by a function $st : P \to \mathbb{R}_+$ denoting the start time of each task, satisfying the precedence and mutual exclusion constraint, as well as the additional *continuity* constraint stating that $st(p) = 0$ for every $p \in U$.

In order to be adaptive we need a *scheduling strategy*, a rule that may induce a different schedule for each 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 actual value of $d(p)$ is revealed only when p terminates.

Definition 9. (State of schedule). A state of a schedule S at time t is $s = (P^f, P^a, \kappa, P^e)$ such that P^f is a downward-closed subset of $(P, \prec)$ consisting of tasks that have terminated (those satisfying $en(p) \leqslant t$), P^a is a set of active tasks currently being executed (those satisfying $st(p) \leqslant t < en(p)$), $\kappa : P^a \to \mathbb{R}_+$ is a function such that $\kappa(p) = t - st(p)$

[9] 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.

[10] Of course, there is a trade-off between what we gain by reducing online computation time and what we pay in terms of offline computation time and in terms of the space needed to store the strategy.

indicates the time elapsed since the activation of p and P^e is the set of enabled tasks, those whose predecessors are in P^f. The set of all possible states is denoted by $\mathcal{S}$.

Definition 10. (Scheduling strategy). A (state-based) scheduling strategy is a function $\sigma : \mathcal{S} \to P \cup \{\bot\}$ such that for every $s = (P^f, P^a, c, P^e)$, $\sigma(s) \in P^e \cup \{\bot\}$ and if $\sigma(s) = p$ then $\mu(p) \neq \mu(p')$ for every $p' \in P^a$.

In other words, a strategy decides at each state whether to do nothing and wait for the next event ($\bot$) or to start executing an enabled task which is not in conflict with any active task. 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 schedule consists of time passage interleaved with 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 the insertion of 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 $S(d, \sigma)$.

Remark. In certain types of games, the optimal strategy may be history-dependent, that is, it will make different decisions at the same state depending on the path through which it has been reached. However in games like those considered in the paper where the cost function is additive, it can be shown that state-strategies (also known as *positional* strategies in Game Theory) are sufficient for optimality.

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. We use $a \dot{-} b$ for $\max\{0, a - b\}$ and $[a, b] \dot{-} c$ for $[a \dot{-} c, b \dot{-} c]$.

Definition 11. (Residual problem). Let $s = (P^f, P^a, \kappa, P^e)$ be a state of a schedule for the problem $\mathcal{J} = (P, M, \prec, \mu, D, U)$. 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

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

Likewise a residual instance d_s is the instance d restricted to $P - P^f$ defined as

$$d_s(p) = \begin{cases} d(p) \dot{-} \kappa(p) & \text{if } p \in P^a, \\ d(p) & \text{otherwise.} \end{cases}$$

Let d be an instance. A strategy σ is *d-future-optimal* if for every instance d' and from every (σ, d')-reachable state s, it produces the optimal schedule for the residual problem $\mathcal{J}_s(d_s)$. If we take d to be the maximal instance, this is exactly the property of the online re-scheduling approach described informally in the previous section.

6. Optimal strategies for timed automata

In this section we show how the problem of finding d-future optimal strategies can be formulated and solved algorithmically using timed automata. The algorithm will be presented in two levels of abstraction. At the higher level, we present a dynamic programming algorithm that computes iteratively a *value function* defined on the state space of the timed automaton. This is the cost-to-go function denoting the length of the shortest path to termination from each configuration. At the more concrete level we explain how this function is represented and computed using a slight modification of the standard backward reachability algorithm for timed automata.

6.1. Modeling

The modeling of the problem with timed automata is similar to Definition 4 with more attention paid to the distinction between *controlled* and *uncontrolled* transitions. The automaton $\mathcal{A}_D^p$ of Fig. 9 models all the possible (isolated) behaviors of a task p with $D(p) = [l, u]$. The *start* transition is controlled and can be initiated by the scheduler any time,

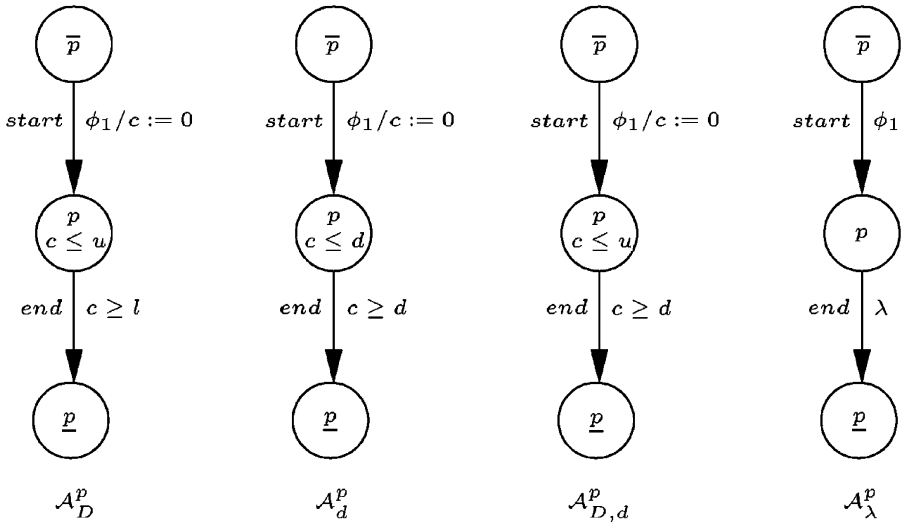

Fig. 9. The generic automaton $\mathcal{A}^p_D$ for a task p such that $D(p) = [l, u]$. The automaton $\mathcal{A}^p_d$ for a deterministic instance d. The automaton $\mathcal{A}^p_{D,d}$ for computing d-future optimal strategies and the automaton $\mathcal{A}^p_\lambda$ for an exponentially distributed duration. Staying conditions for $\overline{p}$ and $\underline{p}$ are *true* and are omitted from the figure.

given that precedence constraints are met. The *end* transition is initiated by the environment and can be taken within $t \in [l, u]$ time after *start*. This uncertainty is modeled using the staying condition $c \leqslant u$ for state p and the transition guard $c \geqslant l$.[11] Composing these automata, as in the deterministic case, gives a global automaton $\mathcal{A}_D$, such that the set of its runs covers all the schedules that are feasible under all possible combinations of strategies and instances.

As a running example consider a simplified version of the example of Section 5 with only one uncertain duration:

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

The automata for the example and their composition appear in Fig. 10. The correspondence between schedule states and reachable configurations is straightforward. Moreover, the residual problem associated with any state of the schedule is represented by the sub-automaton rooted in the corresponding configuration.

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 state-based 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 $(\overline{m}_3, \overline{m}_3)$. Such decisions, as we shall see, may depend also on clock values.

Let Σ be the set of $start_i$ transitions and let Σ_q denote those transitions that are enabled at global state q of the automaton. A scheduling strategy is a partial function $\sigma : Q \times V \to \Sigma \cup \{\bot\}$ such that $\sigma(q, \mathbf{v}) \in \Sigma_q \cup \{\bot\}$, which is defined at least for *every state which is* (d, σ)*-reachable for some instance* d. Let $V_i(q) = \{\mathbf{v} : \sigma(q, \mathbf{v}) = start_i\}$ denote the clock values at which the strategy decides to start task p_i at q, and let $V_\bot(q) = \{\mathbf{v} : \sigma(q, \mathbf{v}) = \bot\}$ be the values at which it decides to wait. Synthesizing the strategy can be seen as eliminating from the automaton the non-determinism on the scheduler side by restricting the guards and staying condition such that at any configuration *only one* transition guard or staying condition holds.

Definition 12. (Strategy automaton). Let $\mathcal{A}_D$ be the automaton describing an uncertain scheduling problem and let $\sigma : Q \times V \to \Sigma \cup \{\bot\}$ be a strategy. The automaton $\mathcal{A}^\sigma_D$, obtained from $\mathcal{A}_D$ by restricting the transition guards of every state q to $V_i(q)$ and intersecting the staying condition with $V_\bot(q)$.

[11] An elegant alternative to using staying condition could be to use timed automata with *deadlines* [44] which are rich enough to express our scheduling problems.

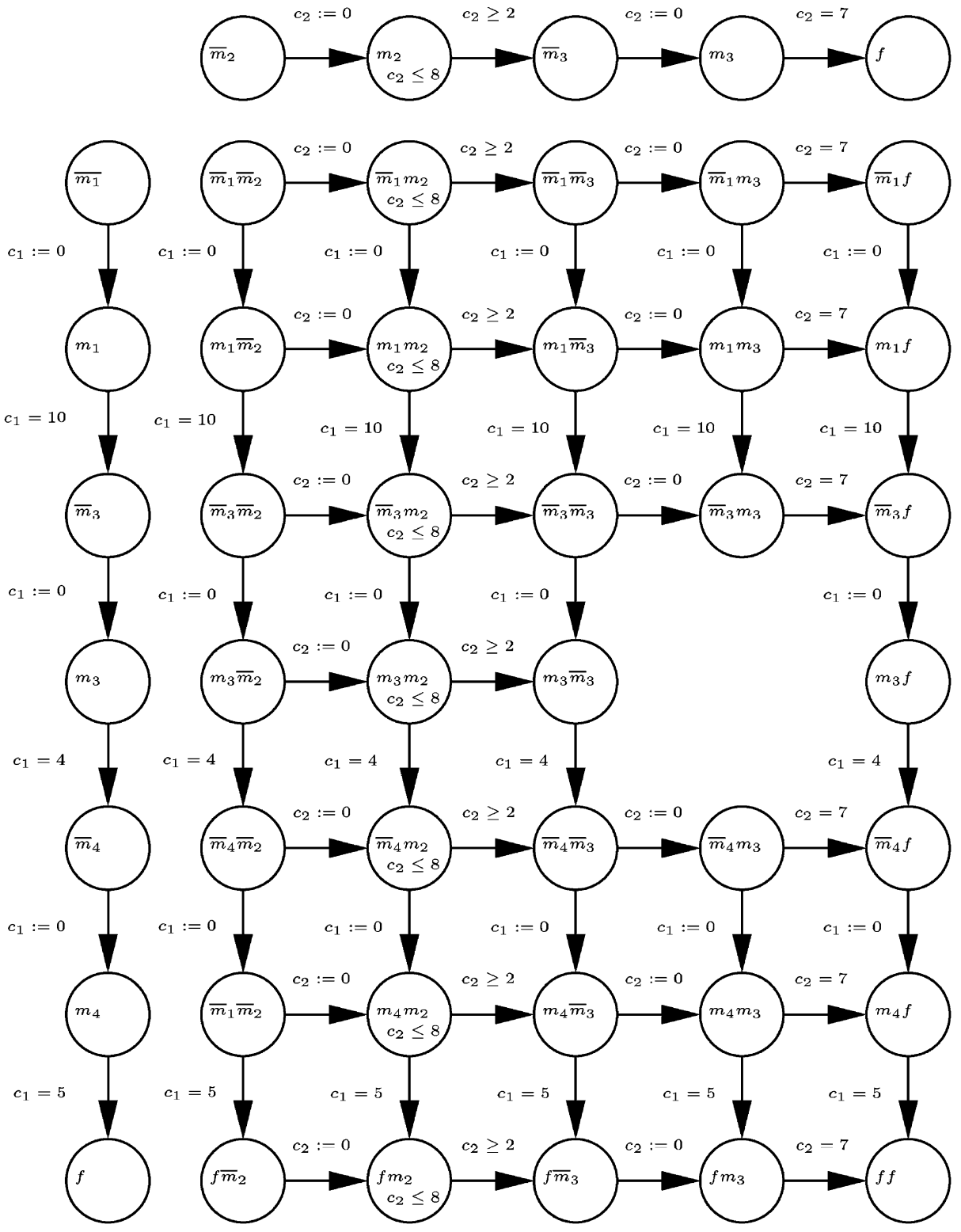

Fig. 10. The global automaton for the job-shop specification. The automata on the left and upper parts of the figure are those of the two jobs.

Note that a priori the sets $V_i(q)$ and $V_\perp(q)$ could have complicated forms which go outside the expressive power of timed automata, but as we shall see (and as was the case in [40,11]), for optimal strategies these sets can be expressed using zones. A strategy σ is *d-future optimal* if from every configuration reachable in $\mathcal{A}_D^\sigma$, it gives the shortest path to the final state (assuming that the remaining uncontrolled transitions are taken according to d). In the sequel we use a simplified form of the definitions and the algorithm of [11] to find such strategies.

6.2. The value function and abstract algorithm

The particularity of d-future optimal strategies where the "current" state (where a decision should be taken) could have been reached by any choice of duration by the environment, while the decision at that state is based on assuming d for the future, forces us to use a slightly modified automaton to do our computation. We will use the automaton $\mathcal{A}_{D,d}^p$ of Fig. 9 to model each task. It can terminate as soon as $c \geqslant d$ but can stay in p until $c = u$. We denote by $\mathcal{A}_{D,d} = (Q, C, I, \Delta, s, f)$ the automaton obtained by composing these automata.

The strategy is obtained as a side effect of computing the *value function* $h : Q \times V \to \mathbb{R}_+$ where $h(q, \mathbf{v})$ 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 satisfies

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

In other words, to compute $h(q, \mathbf{v})$ we should compare all the possibilities to stay some time t in q and then take a transition to some $(q', \mathbf{v}')$. The Bellman principle guarantees that the value associated with such possibility is the sum of t plus the value of $(q', \mathbf{v}')$. Note that h can be written as

$$h(q, \mathbf{v}) = \min\{h_\delta(q, \mathbf{v}), h_\perp(q, \mathbf{v})\},$$

where

$$h_\delta(q, \mathbf{v}) = \min\{h(q', \mathbf{v}') : (q, \mathbf{v} \xrightarrow{0} (q', \mathbf{v}')\}$$

is the value achieved by taking the best enabled transition immediately (non-laziness), and

$$h_\perp(q, \mathbf{v}) = t + h(q', \mathbf{v}')$$

is the value associated with waiting, where t is the minimal distance from $\mathbf{v}$ to a guard of an uncontrolled transition to $(q', \mathbf{v}')$ while assuming instance d (recall that the guards in the automaton on which the computation is performed are of the form $c \geqslant d$).

To understand how this works in our case consider a configuration $(q_0, \mathbf{v}_0) = (p_1, p_2, \overline{p}_3, \overline{p}_4, v_1, v_2, \perp, \perp)$ where two tasks are executing and two tasks are waiting (see Fig. 11). Among the two uncontrolled *end* transitions only one can be taken, namely end_1 if $d_1 - v_1 < d_2 - v_2$ or end_2 otherwise.[12] Suppose that the first case holds, and let $t = d_1 - v_1$. The other transitions can be taken anytime between 0 and t but the non-laziness result, which still holds because we assume a deterministic d-future, tells us that if we take them, we should take them immediately. Hence in this case we have $h(q_0, \mathbf{v}_0) = \min\{d_1 - v_1 + h(q_1, \mathbf{v}_1), h(q_3, \mathbf{v}_3), h(q_4, \mathbf{v}_4)\}$. In another configuration where $d_1 - v_1 > d_2 - v_2$ we should replace $d_1 - v_1 + h(q_1, \mathbf{v}_1)$ by $d_2 - v_2 + h(q_2, \mathbf{v}_2)$. Note that h should be defined also for clock valuations where $v_i > d_i$ but still less than u_i.

The abstract algorithm for computing h works iteratively by letting

$$h_0(q, \mathbf{v}) = \begin{cases} 0 & \text{if } q = f, \\ \infty & \text{otherwise} \end{cases}$$

and then

$$h_{k+1}(q, \mathbf{v}) = \min(\{h_k(q, \mathbf{v})\} \cup \{t + h_k(q', \mathbf{v}') : (q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t\mathbf{1}) \xrightarrow{0} (q', \mathbf{v}')\}),$$

[12] We ignore here the case of equality when two uncontrolled transitions are enabled at exactly the same time. The special structure of our automata guarantees that such transitions commute anyway.

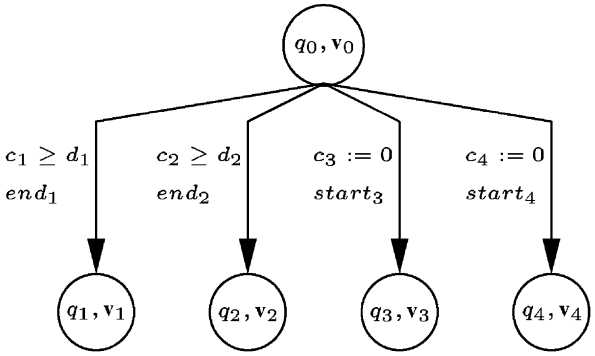

Fig. 11. Computing the value function.

until $h_{k+1} = h_k$. The correctness of this procedure for the more general case of arbitrary timed automata has been proved in [11]. The proof is based on showing that $h_k(q, \mathbf{v})$ is the length of the shortest path (among those that have not more than k transitions) from $(q, \mathbf{v})$ to termination, and that the h_k's range over a class of "nice" functions closely related to the zones used in the verification of timed automata. This class is well-founded and hence the computation of h terminates even for cyclic automata, a fact that we do not need here as h is computed in one sweep through all acyclic paths from the final to the initial state. Note that all such paths have the same number of transitions.

After having computed h, the extraction of a strategy is straightforward: if the optimum of h at $(q, \mathbf{v})$ is obtained via a controlled $start_i$ transition we let $\sigma(q, \mathbf{v}) = start_i$, otherwise we let $\sigma(q, \mathbf{v}) = \perp$. In case when the optimum is obtained via more than one continuation we can define some "tie breaking" rules which prefer, say, waiting over action, and start the task with the least index when there are several candidates.

Before presenting the more concrete version of the algorithm let us illustrate the computation of h on our example. We start with

$$h(f, f, \perp, \perp) = 0,$$
$$h(m_4, f, v_1, \perp) = 5 \dot{-} v_1,$$
$$h(f, m_3, \perp, v_2) = 7 \dot{-} v_2,$$

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 which of m_3, m_4 will terminate first and whether the shorter path goes via (m_4, f) or (f, m_3)

$$\begin{aligned} h(m_4, m_3, v_1, v_2) &= \min \left\{ \begin{array}{l} 7 \dot{-} v_2 + h(m_4, f, v_1 + 7 \dot{-} v_2, \perp), \\ 5 \dot{-} v_1 + h(f, m_3, \perp, v_2 + 5 \dot{-} v_1) \end{array} \right\} \\ &= \min\{5 \dot{-} v_1, 7 \dot{-} v_2\} \\ &= \left\{ \begin{array}{ll} 5 \dot{-} v_1 & \text{if } v_2 \dot{-} v_1 \geqslant 2, \\ 7 \dot{-} v_2 & \text{if } v_2 \dot{-} v_1 \leqslant 2. \end{array} \right. \end{aligned}$$

Note that the corresponding transitions are both uncontrolled *end* transitions and no decision of the scheduler is required in this state.

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

$$\begin{aligned} h(m_1, \overline{m}_3, v_1, \perp) &= \min\{16, 21 \dot{-} v_1\} \\ &= \left\{ \begin{array}{ll} 16 & \text{if } v_1 \leqslant 5, \\ 21 \dot{-} v_1 & \text{if } v_1 \geqslant 5. \end{array} \right. \end{aligned}$$

As for the strategy, one can see that at $(m_1, \overline{m}_3)$ the optimal result is obtained by giving m_3 immediately to J^2 and moving to (m_1, m_3) when $v_1 \leqslant 5$ or by waiting to the termination of m_1, reaching $(\overline{m}_3, \overline{m}_3)$ and then moving to $(m_3, \overline{m}_3)$ if $v_1 \geqslant 5$. 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. Fig. 8(d) shows that, indeed, the two choices coincide in performance when $v_1 = 5$.

6.3. The concrete algorithm

We now describe how h is computed using a standard backward reachability algorithm that works on sets, not on functions. Let Θ be an upper bound on the length of the worst-case optimal schedule, for example the sum of the maximal durations of all tasks. Instead of computing h directly we compute the set

$$R = \{(q, \mathbf{v}, t) : \Theta - t \geqslant h(q, \mathbf{v}) \wedge t \geqslant 0\}.$$

Note, that R contains exactly the same information as h and, in particular, h can be reconstructed from R:

$$h(q, \mathbf{v}) = \min\{\Theta - t : (q, \mathbf{v}, t) \in R\}.$$

Moreover, since $h(q, \mathbf{v}) \leqslant \Theta$ for any $(q, \mathbf{v})$ which is forward reachable in $\mathcal{A}_D$, any such $(q, \mathbf{v})$ is a projection of some point $(q, \mathbf{v}, t) \in R$. Consequently, computing R amounts to computing the strategy for every reachable configuration.

From the definitions of h and R, a triple $(q, \mathbf{v}, t)$ belongs to R iff one can reach the final state $(f, \perp)$ of the automaton from $(q, \mathbf{v})$ within $\Theta - t$ time, assuming instance d for all tasks that have not terminated in $(q, \mathbf{v})$. Hence the set R can be characterized in terms of reachability as follows. Let $\mathcal{A}'_{D,d}$ be the auxiliary automaton obtained by augmenting $\mathcal{A}_{D,d}$ with a clock T which is never reset to zero (as in Section 4) and by adding the constraint $T < \Theta$ to the staying condition of every state (to avoid divergence of T). The following result gives a useful characterization of R.

Lemma 1. *A configuration $(q, \mathbf{v}, t) \in R$ iff the state $(f, \perp, \Theta)$ is reachable from $(q, \mathbf{v}, t)$ in $\mathcal{A}'_{D,d}$.*

Proof. If there is a run in $\mathcal{A}'_{D,d}$, from $(q, \mathbf{v}, t)$ to $(f, \perp, \Theta)$, then, by the definition of the auxiliary clock T, this run is of duration $\Theta - t$ and $(q, \mathbf{v}, t) \in R$. Conversely, if $(q, \mathbf{v}, t) \in R$, then there exists $t' \geqslant t$ and a run of $\mathcal{A}'_{D,d}$ from $(q, \mathbf{v}, t')$ to $(f, \perp, \Theta)$ of duration $\Theta - t'$. By subtracting $(t' - t)$ from T on both sides of the run we obtain a run from $(q, \mathbf{v}, t)$ to $(f, \perp, \Theta - (t' - t))$ which can be extended, via idling for $t' - t$ time in f, into a run of length $\Theta - t$. □

Hence, the set R can be obtained by the standard backward reachability algorithm for timed automata, and has a form of a finite union of zones. For completeness we instantiate the backward reachability algorithm for this case.

We recall some commonly-used definitions in the verification of timed automata [33]. A *zone* is a subset of V consisting of points satisfying a conjunction of inequalities of the form $c_i - c_j \geqslant k$ or $c_i \geqslant k$. 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 $\{(q, \mathbf{v}) : \mathbf{v} \in Z\}$. Zones and symbolic states are closed under various operations including the following:

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

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

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

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

- 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:

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

The result can be represented as a set of symbolic states.

Algorithm 2 is based on the standard backward reachability algorithm for timed automata. It starts with the final state of $\mathcal{A}'$ (with $T = \Theta$) in a waiting list and outputs 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.

Algorithm 2 (*Backward reachability for timed automata*).
Waiting $:= \{(f, \{(\bot, \Theta)\}, \emptyset)\}$;
Explored $:= \emptyset$;
while *Waiting* $\neq \emptyset$ **do**
Pick $(q, Z, q'') \in$ *Waiting*;
For every $(q', Z') \in Pre(q, Z)$;
Insert (q', Z', q) *into Waiting*;
Move (q, Z, q'') *from Waiting to Explored*
end
$E :=$ *Explored*

The backward reachable set of states R is related to the set of triples E as follows:

$$(q, \mathbf{v}, t) \in R \quad \Leftrightarrow \quad \exists Z, q' : (\mathbf{v}, t) \in Z \wedge (q, Z, q') \in E.$$

For implementing the strategy it is convenient to use the set of triples E. Whenever a transition to $(q, \mathbf{v})$ is taken during the execution we look at all the symbolic states with discrete state q and find by which tuple (q, Z, q') the minimum

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

is obtained. 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. This concludes the major contribution of this paper, an algorithm for computing d-future optimal strategies for the problem of job-shop scheduling under uncertainty.

Result 1 (*Computing d-future optimal strategies*). *The problem of finding d-future optimal strategies for job-shop scheduling problem under uncertainty is solvable using timed automata reachability algorithms.*

6.4. Experimental results

We have implemented Algorithm 2 using the zone library of Kronos/IF [19,20], as well as the hole-filling strategy. As a benchmark we took the following problem with 4 jobs and 6 machines:

J^1 : $(m_2, 34), (m_4, [21, 54]), (m_3, 74), (m_5, [6, 26]), (m_1, 5), (m_6, 43)$,
J^2 : $(m_2, 24), (m_5, [13, 28]), (m_1, 53), (m_3, 8), (m_6, [16, 23]), (m_4, 45)$,
J^3 : $(m_6, [35, 75]), (m_5, 14), (m_3, [8, 15]), (m_1, 31), (m_2, 24), (m_4, 6)$,
J^4 : $(m_1, [12, 42]), (m_3, [25, 32]), (m_6, 15), (m_4, 42), (m_5, 62), (m_2, 18)$.

The static worst-case optimal schedule for this problem is 268. We have applied Algorithm 1 to find d-future optimal strategies based on two instances that correspond, respectively, to "optimistic" and "pessimistic" predictions. For every p such that $D(p) = [l, u]$ they are defined as $d_{\min}(p) = l$ and $d_{\max}(p) = u$. In addition we have synthesized a hole-filling strategy based on these two instances. We have generated 100 random instances with task durations drawn uniformly from each $[l, u]$ interval, and compared the results of the abovementioned strategies with an optimal clairvoyant scheduler [13] that knows d in advance, and with the static worst-case scheduler. It turns out that the static schedule is, in the average, longer than the optimum by 12.54%. The hole filling strategy deviates from the optimum by 4.90% (for optimistic prediction) and 4.44% (for pessimistic prediction). Our strategy produces schedules that are longer than the optimum by 1.40% and 1.14%, respectively.

Having demonstrated that adaptive strategies can lead to more efficient schedules, the question of scaling-up the results to larger problems remains. Currently the computation of a strategy for the 4×6 example takes around 10 min

[13] In the domain of *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* of the algorithm.

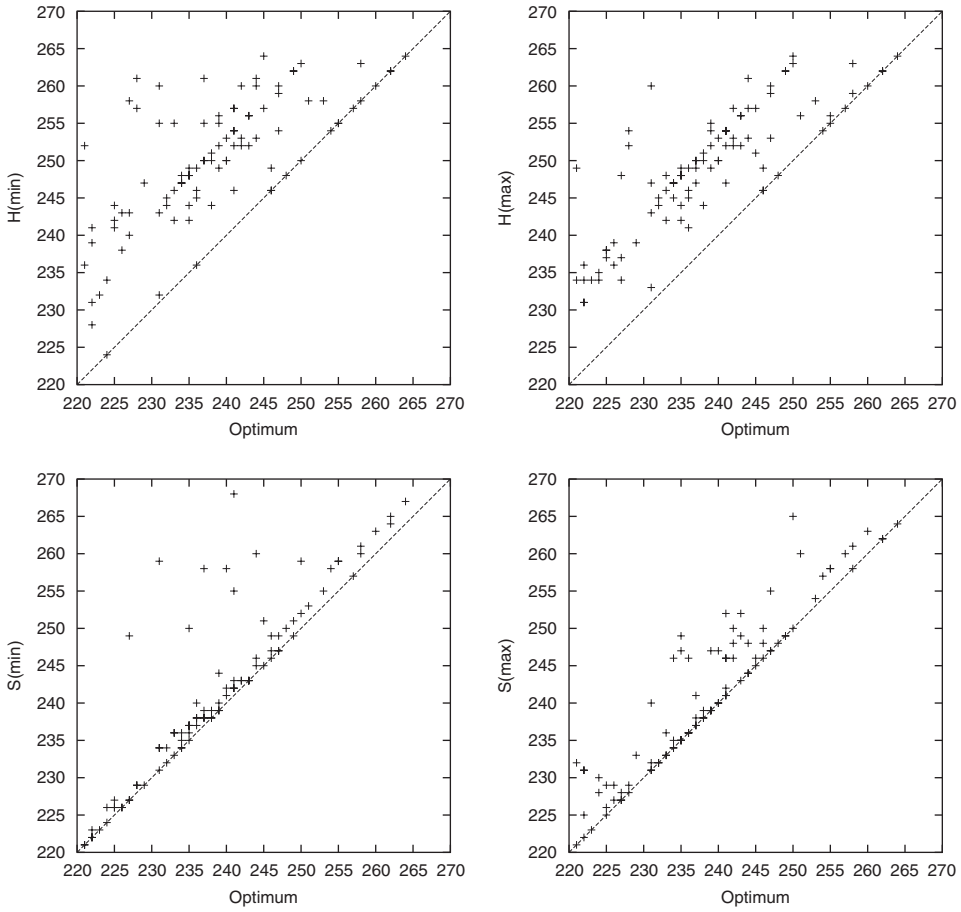

Fig. 12. The quality of schedules produced by the hole-filling (H) strategy and the d-future optimal strategy (S) using optimistic and pessimistic predictions. Each instance is drawn as a point (x, y) on the plane with x indicating the length of the optimal schedule and y—the length of the schedule produced by the corresponding strategy.

and there is not much hope to go significantly beyond this size using exhaustive backward reachability.[14] For the deterministic case, we have shown that much larger problems can be solved using forward reachability algorithms that do not use zones and that can employ intelligent search strategies combined with heuristics to prune the search space. Apparently this is not the case for uncertain problems where exhaustive backward computations on zones seems unavoidable. The reason is that, unlike deterministic problems where the scheduler alone determines the set of reachable states, under uncertainty the environment can lead the automaton to a large portion of the discrete state space and to uncountably-many clock valuations. The strategy needs to be defined for all of them. As one can see from the results, the more conservative and sub-optimal hole-filling strategy produces reasonable schedules with much more modest computation—it just computes the optimal strategy for a deterministic problem and can do it for significantly larger systems.

7. Probabilistic uncertainty

In this section we describe briefly how optimal scheduling under *probabilistic temporal uncertainty* can be formulated and solved using similar techniques. We assume uncertainties in task durations to be exponentially distributed, that is,

[14] The computation of the strategy for exponential distribution described in the next section is much faster because it involves no clocks and zones, but it is subject to the same type of state explosion.

we associate with each task a parameter λ such that its duration d is a random variable satisfying

$$P(d \geqslant T) = \mathrm{e}^{-\lambda T}.$$

A scheduling problem with n tasks defines a probability distribution over the space of instances $\mathbb{R}_+^n$. A scheduling strategy is, as before, a mechanism for deciding at every point in time whether to start an enabled task or to wait. A strategy together with an instance determines the length of the obtained schedule and we look for a strategy that minimizes the *expected value* (over all instances) of this length.

The automaton $\mathcal{A}_\lambda$ for modeling a task appears in Fig. 9. It is a mixture of a non-deterministic automaton and a continuous time Markov chain. The decision when to make the transition from $\overline{p}$ to p is to be made by the scheduler and is *not* probabilistically distributed. Hence, before the construction of the scheduler we cannot assign probabilities to the runs of the automaton, which are of the form

$$\overline{p} \xrightarrow{r} \overline{p} \xrightarrow{0} p \xrightarrow{d} p \xrightarrow{0} \underline{p} \xrightarrow{\infty},$$

where r is the time chosen by the scheduler to wait before starting p. The product automaton obtained for the scheduling problem has the same discrete structure as for the deterministic or non-deterministic versions, with the λ-labelled *end* transition interpreted probabilistically. For the scheduler to decide in each state whether to start a task or to wait, it has to compare the costs of active actions with that of waiting. There are two major differences compared to the non-deterministic case:

(1) The exponential distribution is memoryless, which means that the probability that an *end* transition is taken at a given state does not depend on the time already spent in this state. [15] Hence an optimal strategy depends only on discrete states and does not need clocks.
(2) If the scheduler decides to wait in a state where two or more tasks are active, the identity of the task that will terminate first as well as its duration are defined probabilistically (this is called "race analysis" in the Markovian jargon).

The optimal strategy is found by a variant of value iteration with a function $h : Q \to \mathbb{R}_+$ be a function such that $h(q)$ is the best achievable expected time from q to the final state f. By definition, $h(f) = 0$ and its value for the other states is computed backwards as follows.

Let q be a state having k outgoing *end* transitions leading to states $q_1, \ldots, q_k$ with parameters $\lambda_1, \ldots, \lambda_k$, respectively, and l outgoing *start* transitions leading to states $q'_1, \ldots, q'_l$, respectively. As before h can be written as

$$h(q) = \min\{h_\delta(q), h_\perp(q)\},$$

where $h_\delta(q)$ is the value of immediate action which is $\min\{h(q'_1), \ldots, h(q'_l)\}$ and $h_\perp$, the value of waiting is given by

$$h_\perp(q) = d + \sum_{j=1}^{k} \gamma_j \cdot h(q_j),$$

where d is the expected duration (over all instances) of staying in q and γ_j is the probability that the transition to q_j will win the race. These are computed as

$$d = \frac{1}{\sum_{a=1}^{k} \lambda_a} \quad \text{and} \quad \gamma_j = \frac{\lambda_j}{\sum_{a=1}^{k} \lambda_a}.$$

This solves to optimal scheduling to this (unexplored, to the best of our knowledge) class of continuous-time Markov decision processes.

[15] This property is a source for both the analytic simplicity of this distribution as well as its modest relevance to certain real-world situations.

8. 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 controlled transitions (and hence make the necessary choices) so that a property is satisfied. If we add a quantitative dimension (in this case, the length or cost of the run), 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 by Wong-Toi and Hoffmann [45] who proved decidability of controller synthesis for timed automata. An algorithm for safety controller synthesis for timed automata, based on operation on zones was first reported in [40] and later in [12], where an example of a simple scheduler was given. This algorithm is a generalization of the verification algorithm for timed automata [33,7] used in Kronos [46,19]. In these and other works on treating scheduling problems as synthesis problems for timed automata, such as [6], the emphasis was on yes/no properties, such as the existence of a feasible schedule, respecting additional constraints such as deadlines, in the presence of an uncontrolled adversary.

A transition toward quantitative evaluation criteria was made already in [23] where timed automata were used to compute bounds on delays in real-time systems and in [21] 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 [11] in which the following problem has been shown to be decidable: "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". The result of [11], achieved using value iteration on h, is very general but has never been implemented. Our algorithm for scheduling under uncertainty can be seen as an instantiation of this algorithm for a special class of timed automata that model scheduling problems.

Around the same time, in the framework of the verification of hybrid systems (VHS) project, a simplified model of a steel plant was presented as a case-study [17]. The model had more features than the job-shop scheduling problem such as upper-bounds on the time between steps, transportation problems, etc. Fehnker proposed a timed automaton model of this plant from which feasible schedules could be extracted [27]. This work inspired us to find a systematic connection between classical scheduling problems and timed automata [38], upon which this work is based. 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 [43].

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 state. The possibility of guiding the search for optimal paths in timed automata was first investigated in [14] where it was applied to several classes of examples, including the job-shop problems.

In [42] 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 algorithms based on forward reachability was introduced. A recent generalization of the shortest path problem was investigated by Behrmann et al. [15] and Alur et al. [10]. In this model there is a *different* price for staying in any state and the total cost associated with run evolves in different rates along the path. It has been proved that the problem of finding the path with the minimal cost is computable. In [5] the results of Section 4 were generalized to scheduling with preemption, while in [3] the case of precedence constraints that do not decompose into chains was treated. Yet another recent application of timed automata to scheduling can be found in [28] where the question of schedulability of preemptible non-periodic tasks under deadline constraints is addressed.

Let us mention briefly some work on scheduling under uncertainty which is not based on automata. Interested readers may consult the recent survey [24]. The fact that uncertainty may arise due to various sources (machine breakdown, unexpected arrival of new orders, modification of existing orders) is well-known to practitioners but the number of scientific publications devoted to this problem is relatively small (and most of them have the flavor of AI planning rather than Operations Research). In [34] it is observed that a schedule which is determined to be optimal prior to its execution is optimal only to the degree that the real world behaves as expected during execution and that a new model of scheduling is needed. In a study of the job-shop problem [41] the authors claim that the dynamic characteristics of some real-world scheduling environment render the bulk of existing solution approaches unusable when applied

to practical problems. In this paper the authors criticize existing job-shop scheduling research by saying: "The static problem definition is so far removed from job-shop reality that perhaps a different name for the research should be considered."

Most of the research dealing with uncertainty has been carried out within the last few years and the main focus was about the existence of feasible schedules satisfying some constraints rather than on optimization. It is clear that some notion of schedule *robustness* is needed in order to assess solutions that have to cope with uncertainty but there is no agreement on the formal definition of this robustness. In general there are two approaches to deal with uncertainty: *pro-active* and *reactive* scheduling. Pro-active techniques create robust schedules that do not need to be modified during the execution while reactive scheduling involves a revision of the schedule when an unexpected event occurs. We mention briefly a pro-active technique (redundancy-based) and a reactive one (contingent scheduling).

Redundancy-based techniques account for uncertainty by inserting some form of redundancy, typically extra time or additional backup resources, into the schedule so that unexpected events during execution can be dealt with. Examples of such methods are fault tolerant real-time scheduling [32,31], slack-based protection [37] and temporal protection [22,30]. These techniques are close in spirit to the static worst-case strategy we have described.

Contingent scheduling techniques attempt to anticipate disruptive events and generate multiple schedules (or schedule fragments) which respond to these events. This is all done a priori so that at execution time a set of schedules is available and the scheduler can switch between them as events occur. This technique was applied in [26] to solve the telescope observation problem, a one machine problem where activities have uncertain durations, and these uncertainties can lead to schedule breakage. This approach can be viewed as an ad hoc version of strategy synthesis and its extension to systems with multiple resources suffers from combinatorial explosion.

Another popular approach for treating planning under uncertainty is to model the scheduling problem as a (discrete time) Markov decision process, and synthesize an average-case optimal strategy, see [18] for a survey. While this approach is natural for representing discrete uncertainty (for example, a machine breaks down with some probability) it is not yet clear how they apply it effectively to temporal uncertainty and how to cope with state explosion in general.

9. Conclusions and future work

We have suggested a novel application of timed automata, namely for solving optimal job-shop scheduling problems. We believe that the insight gained from this point of view will contribute both to scheduling and to the study of timed automata. We have demonstrated that the performance of automaton-based methods is not inferior to other methods developed within the last three decades and have shown how they can be used to synthesize adaptive scheduling strategies for problem with uncertain task durations.

Future research should extend the model toward more complex situations including cyclic tasks, uncertainty in task arrival times, non-monotonic timing constraints and logical dependencies among tasks. On the algorithmic side, an adaptation of forward search algorithms on *game graphs* to timed automata may lead to more efficient algorithms for scheduling under uncertainty. Yet another interesting question is how to adapt this theory to *resource-bounded* schedulers which have to control a fast environment without access to huge tables and with a restricted utilization of clocks.

Acknowledgements

We are grateful to Marius Bozga for his valuable help in the implementation of the algorithms. This work benefitted from interactions with other members of Veriamg and partners in the VHS and AMETIST projects, in particular Kim Larsen, Peter Niebert, Stavros Tripakis, Sergio Yovine, Joseph Sifakis, Ed Brinksma and Sebastian Engell. Comments by anonymous referees improved significantly the rigor of this paper.

References

[1] Y. Abdeddaïm, Scheduling with timed automata, Ph.D. Thesis, INPG, Grenoble, 2002.

[2] Y. Abdeddaı¨m, E. Asarin, O. Maler, On optimal scheduling under uncertainty, in: Proc. TACAS'03, Lecture Notes in Computer Science, vol. 2619, Springer, Berlin, 2003, pp. 240–255.

[3] Y. Abdeddaïm, A. Kerbaa, O. Maler, Task graph scheduling using timed automata, in: Proc. FMPPTA'03, 2003.
[4] Y. Abdeddaïm, O. Maler, Job-shop scheduling using timed automata, in: Proc. CAV'01, Lecture Notes in Computer Science, vol. 2102, Springer, Berlin, 2001, pp. 478–492.
[5] Y. Abdeddaï¨m, O. Maler, Preemptive job-shop scheduling using stopwatch automata, in: Proc. TACAS'02, Lecture Notes in Computer Science, vol. 2280, Springer, Berlin, 2002, pp. 113–126.
[6] K. Altisen, G. Goessler, A. Pnueli, J. Sifakis, S. Tripakis, S. Yovine, A framework for scheduler synthesis, in: Proc. RTSS'99, IEEE, New York, 1999, pp. 154–163.
[7] R. Alur, C. Courcoubetis, D.L. Dill, Model checking in dense real time, Inform. and Comput. 104 (1993) 2–34.
[8] R. Alur, C. Courcoubetis, T.A. Henzinger, Computing accumulated delays in real-time systems, Formal Methods System Design 11 (1997) 137–155.
[9] R. Alur, D.L. Dill, A theory of timed automata, Theoret. Comput. Sci. 126 (1994) 183–235.
[10] R. Alur, S. La Torre, G.J. Pappas, Optimal paths in weighted timed automata, in: Proc. HSCC'01, Lecture Notes in Computer Science, vol. 2034, Springer, Berlin, 2001, pp. 49–64.
[11] E. Asarin, O. Maler, As soon as possible: time optimal control for timed automata, in: Proc. HSCC'99, Lecture Notes in Computer Science, vol. 1569, Springer, Berlin, 1999, pp. 19–30.
[12] E. Asarin, O. Maler, A. Pnueli, Symbolic controller synthesis for discrete and timed systems, Hybrid Systems II, Lecture Notes in Computer Science, vol. 999, Springer, Berlin, 1995, pp. 1–20.
[13] E. Asarin, O. Maler, A. Pnueli, J. Sifakis, Controller synthesis for timed automata, in: Proc. IFAC Sympos. System Structure and Control, Elsevier, Amsterdam, 1998, pp. 469–474.
[14] G. Behrmann, A. Fehnker T.S. Hune, K.G. Larsen, P. Pettersson, J. Romijn, Efficient guiding towards cost-optimality in UPPAAL, in: Proc. TACAS'01, Lecture Notes in Computer Science, vol. 2031, Springer, Berlin, 2001, pp. 174–188.
[15] G. Behrmann, A. Fehnker T.S. Hune, K.G. Larsen, P. Pettersson, J. Romijn, F.W. Vaandrager, Minimum-cost reachability for linearly priced timed automata, in: Proc. HSCC'01, Lecture Notes in Computer Science, vol. 2034, Springer, Berlin, 2001, pp. 147–161.
[16] B. Berthomieu, M. Diaz, Modeling and verification of time dependent systems using time petri nets, IEEE Trans. Software Eng. 17 (1991) 259–273.
[17] R. Boel, G. Stremersch, VHS case study 5: modelling and verification of scheduling for steel plant at SIDMAR, Draft, 1999.
[18] C. Boutilier, T. Dean, S. Hanks, Decision-theoretic planning: structural assumptions and computational leverage, J. Artif. Intell. Res. 11 (1999) 1–94.
[19] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, S. Yovine, Kronos: a model-checking tool for real-time systems, in: Proc. CAV'98, Lecture Notes in Computer Science, vol. 1427, Springer, Berlin, 1998.
[20] M. Bozga, S. Graf, L. Mounier, IF-2.0: a validation environment for component-based real-time systems, in: Proc. CAV'02, Lecture Notes in Computer Science, vol. 2404, Springer, Berlin, 2002.
[21] S. Campos, E. Clarke, W. Marrero, M. Minea, H. Hiraishi, Computing quantitative characteristics of finite-state real-time systems, in: Proc. RTSS'94, IEEE, New York, 1994.
[22] W.Y. Chiang, M.S. Fox, Protection against uncertainty in a deterministic schedule, in: Proc. Fourth Internat. Conf. Expert Systems and Leading Edge in Production and Operations Management, 1990, pp. 184–197.
[23] C. Courcoubetis, M. Yannakakis, Minimum and maximum delay problems in real-time systems, in: Proc. CAV'91, Lecture Notes in Computer Science, vol. 575, Springer, Berlin, 1991, pp. 399–409.
[24] A.J. Davenport, J.Ch. Beck, Managing uncertainty in scheduling: a survey, 2000, preprint.
[25] C. Daws, S. Yovine, Reducing the number of clock variables of timed automata, in: Proc. RTSS'96, IEEE, New York, 1996, pp. 73–81.
[26] M. Drummond, J. Bresina, K. Swanson, Just-in-case scheduling, in: Proc. AAAI-94, 1994.
[27] A. Fehnker, Scheduling a steel plant with timed automata, in: Proc. RTCSA'99, 1999.
[28] E. Fersman, P. Pettersson, W. Yi, Timed automata with asynchrounous processes: schedulability and decidability, in: Proc. TACAS'02, Lecture Notes in Computer Science, vol. 2280, Springer, Berlin, 2002, pp. 67–82.
[29] H. Fisher, G.L. Thompson, Probabilistic learning combinations of local job-shop scheduling rules, in: J.F. Muth, G.L. Thompson (Eds.), Industrial Scheduling, Prentice-Hall, Englewood Cliffs, NJ, 1963, pp. 225–251.
[30] H. Gao, Building robust schedules using temporal protection, Master's Thesis, Department of Industrial Engineering, University of Toronto, 1995.
[31] S. Ghosh, Guanranteeing fault-tolerance through scheduling in real-time systems, Ph.D. Thesis, University of Pittsburgh, 1996.
[32] S. Ghosh, R. Melhem, D. Mosse, Enhancing real-time schedules to tolerate transient faults, in: Proc. RTSS'95, 1995, pp. 120–129.
[33] T.A. Henzinger, X. Nicollin, J. Sifakis, S. Yovine, Symbolic model-checking for real-time systems, Inform. and Comput. 111 (1994) 193–244.
[34] D.W. Hildum, Flexibility in a knowledge-based system for solving dynamic resource constrained scheduling problems, Ph.D. Thesis, Department of Computer Science, University of Massachusetts, 1994.
[35] A.S. Jain, S. Meeran, Deterministic job-shop scheduling: past, present and future, European J. Oper. Res. 113 (1999) 390–434.
[36] K.G. Larsen, P. Pettersson, W. Yi, UPPAAL in a nutshell, Software Tools for Tech. Transfer 1/2, 1997.
[37] V.J. Leon, S.D. Wu, R.H. Storer, Robustness measures and robust scheduling for job shops, IIE Trans. 26 (1994) 32–43.
[38] O. Maler, On the problem of task scheduling, Draft, February 1999.
[39] O. Maler, On optimal and sub-optimal control in the presence of adversaries, in: Proc. WODES'04, 2004, pp. 1–12.
[40] O. Maler, A. Pnueli, J. Sifakis, On the synthesis of discrete controllers for timed systems, in: Proc. STACS'95, Lecture Notes in Computer Science, vol. 900, Springer, Berlin, 1995, pp. 229–242.
[41] K.N. McKay, F.R. Safayeni, J.A. Buzacott, Job-shop scheduling theory: what is relevant?, Interfaces 18 (1998) 84–90.
[42] P. Niebert, S. Tripakis, S. Yovine, Minimum-time reachability for timed automata, IEEE Mediteranean Control Conf., 2000.

[43] P. Niebert, S. Yovine, Computing optimal operation schemes for chemical plants in multi-batch mode, European J. Control 7 (2001) 440–453.
[44] J. Sifakis, S. Yovine, Compositional specification of timed systems, in: Proc. STACS'96, Lecture Notes in Computer Science, vol. 1046, Springer, Berlin, 1996, pp. 347–359.
[45] H. Wong-Toi, G. Hoffmann, The control of dense real-time discrete event systems, Technical Report STAN-CS-92-1411, Stanford University, 1992.
[46] S. Yovine, Kronos: a verification tool for real-time systems, Internat. J. Software Tools for Tech. Transfer 1 (1997).