On Optimal Scheduling under Uncertainty[⋆]

Yasmina Abdeddaïm, Eugene Asarin, and Oded Maler

VERIMAG, Centre Equation, 2, av. de Vignate 38610 Gières, France
{Yasmina.Abdeddaim,Eugene.Asarin,Oded.Maler}@imag.fr

Abstract. In this work we treat the problem of scheduling under two types of temporal uncertainty, set-based and probabilistic. For the former we define appropriate optimality criteria and develop an algorithm for finding optimal scheduling strategies using a backward reachability algorithm for timed automata. For probabilistic uncertainty we define and solve a special case of continuous-time Markov Decision Process. The results have been implemented and were applied to benchmarks to provide a preliminary assessment of the merits of each approach.

1 Introduction

The problem of evaluating or optimizing the performance of an open reactive system, that is, a system that interacts with an external environment, raises some serious conceptual problems. Given such a 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 the system performance. Several approaches to this problem are commonly used:
1) *Worst-case*: the system is evaluated according to its worst behavior.
2) *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.
3) *Nominal-case*: the system is evaluated according to its performance with respect to 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 incorrect if one of its behaviors violates 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.[1]

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

[⋆] This work was partially supported by the European Community Projects IST-2001-35304 AMETIST (Advanced Methods for Timed Systems), http://ametist.cs.utwente.nl and IST-2001-33520 CC (Control and Computation), http://www.dii.unisi.it/~hybrid/cc/

[1] 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 to counteract all conceivable traffic jams.

graceful degradation in moments of extreme pressure from the environment. 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 in the system life-time and the performance of other behaviors varies "continuously" with the distance from the nominal one. This approach is widely used in control theory.

From a computational standpoint the nominal approach is the easiest because when d is fixed the system is closed and $S(d)$ can be computed by simple simulation. 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" with all instances in order to find that behavior. 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 generally[2] the most difficult because not only do we need to explore all behaviors but also keep track of their probabilities in order to compute the overall evaluation of the system.

In this work we treat the problem of job-shop scheduling under temporal uncertainty. The system to be designed is a scheduler, i.e. a mechanism that controls the allocation of resources to competing tasks. The environment consists of tasks, all known in advance, that need to be executed on certain machines while satisfying some ordering constraints. The only source of uncertainty is the *duration* of the tasks which is known to be bounded within an interval of the form $[l, u]$. Alternatively, the duration of each task can be given as a continuous random variable. Each *instance* (also called *realization*) of the environment consists of selecting a number $d \in [l, u]$ for every task. The behavior induced by the scheduler on this instance is evaluated according to the length of the schedule, i.e. the termination time of the last task executed.

As a running example consider two jobs

$$J_1 = (m_1, 10) \prec (m_3, [2, 4]) \prec (m_4, 5) \qquad J_2 = (m_2, [2, 8]) \prec (m_3, 7)$$

with the intended meaning that J_1 has to use m_1 for 10 time, then m_3 for a period between 2 and 4 time, then m_4 for 5, etc. In this example the only resource under conflict is m_3 and the order of its usage is the only decision the scheduler needs to take. 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 $d = (d_1, d_2) \in [2, 4] \times [2, 8]$. 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_1, for example, is known *only after the termination of* m_2.

Each instance defines a deterministic scheduling problem admitting one or more optimal solutions. Such a solution specifies the start time of every task. Figure 1-(a) depicts optimal schedules for the instances $(8, 4)$, $(8, 2)$ and $(4, 4)$. Of course, such an optimal schedule can only be generated by a *clairvoyant* scheduler who knows the whole instance in advance.

For this type of problems, worst-case optimization reduces to nominal-case because there is one specific instance, namely the one where each task terminates the latest possible, such that the performance of any scheduler on this instance will be at least

[2] At least when the approach is applied naively without using additional mathematical information that can simplify the solution in some special cases.

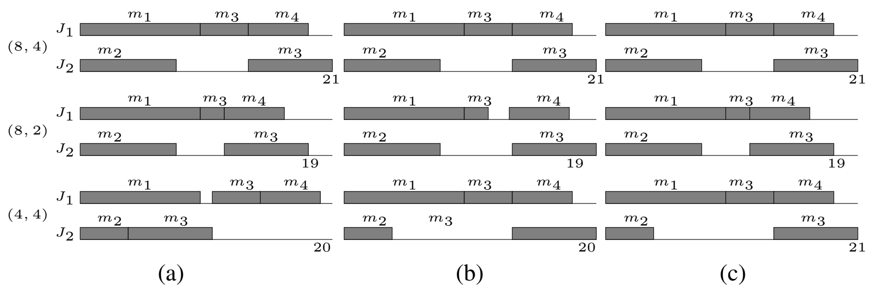

Fig. 1. (a) Optimal schedules for three instances. For the first two the optimum is obtained with $J_1 \prec J_2$ on m_3 while for the third — with $J_2 \prec J_1$; (b) A static schedule based on the worst instance $(8, 4)$. It gives the same length for all instances; (c) The behavior of a hole filling strategy based on instance $(8, 4)$.

as bad as on any other instance. This trivializes the problem of worst-case optimization because we can do the following: find an optimal schedule for the worst instance, extract the start time for each task and stick to the schedule regardless of the actual instance. The behavior of a static scheduler for our example, based on instance $(8, 4)$ is depicted in Figure 1-(b), and one can see that it is rather wasteful for other instances. Intuitively we will prefer a smarter adaptive scheduler that reacts to the evolution of the environment and uses additional information revealed during the execution of the schedule. This is the essential difference between a schedule (a plan, 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 for how long) and decides accordingly what to do. When there is no uncertainty, the scheduler knows exactly what will be the state at every time instant, so the strategy can be reduced to a simple assignment of start times to tasks.

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", we extract from it only the qualitative information, namely the order in which conflicting tasks utilize each resource. In our example the optimal schedule for the worst instance $(8, 4)$ is associated with the ordering $J_1 \prec J_2$ on m_3. Then, during execution, we start every task as soon as its predecessors have terminated, provided that the ordering is not violated (a similar strategy was used in [NY00] and probably elsewhere). As Figure 1-(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 $J_1 \prec J_2$ 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 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 it will be occupied.

While the hole filling strategy can be shown to be optimal for all those instances whose optimal schedule has the same ordering as that of the nominal instance, it is not good for instances such as $(4, 4)$, where a more radical form of adaptiveness is required. If we look at the optimal schedules for $(8, 4)$ and $(4, 4)$ (Figure 1-(a)) we see that the decision whether or not to execute the second task of J_2 is done in both cases in the same qualitative state, namely m_1 is executing and m_2 has terminated. The only difference is in the 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 automata models, is represented by clock values.

Consider the following approach: initially we find an optimal schedule for some nominal instance. During the execution, whenever a task terminates (before or after the time it was assumed to) we re-schedule the "residual" problem, assuming nominal values for the tasks that have not yet terminated. In our example, we first build an optimal schedule for $(8, 4)$. If task m_2 in J_2 has terminated after 4 time we have the residual problem

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

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 start m_3 of J_2. Likewise if m_2 terminates at 8 we have

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

and the optimal schedule consists of waiting for the termination of m_1 and then starting m_3 of J_1. The property of the schedules obtained this way, is that at any moment in the execution they are optimal with respect to the nominal assumption concerning the *future*. 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.

This approach involves a lot of *on-line* computation, namely solving a new scheduling problem each time a task terminates. The alternative approach that we propose in this paper is based on expressing the scheduling problem using timed automata and synthesizing a controller *off-line*. In this framework [AMPS98,AM99,AGP$^+$99] a strategy is a function from states and clock valuations to controller actions (in this case starting tasks). After computing such a strategy and representing it properly, the execution of the schedule may proceed while keeping track of the state of the corresponding automaton. Whenever a task terminates, the optimal action is quickly computed from the strategy look-up table and the results are identical to those obtained via on-line re-scheduling. Of course, there is a trade-off between what we gain in reducing on-line computation time and what we pay in terms of the time and space needed to compute and store the strategy, but this is outside the scope of the current paper.

The rest of the paper is organized as follows. In Section 2 we describe the model and characterize the properties of the dynamic schedulers we want to compute. In section 3 we show how to model the problem using timed automata. The algorithm for synthesizing optimal strategies is described in Section 4 along with its implementation using the zone library of Kronos. In Section 5 we formulate and solve the same scheduling problem where tasks durations are known to be distributed probabilistically. Section 6 concludes the paper with a brief review of the experimental results and future directions. Due to time and space limitation, large parts of the paper are written at an informal intuitive level. Readers interested in more precise definitions or in the details of the experimental results may consult the expanded version of this paper.[3]

2 The Model

We will use a formulation which is slightly more general than the standard job-shop problem by allowing a partial-order relation between tasks. We denote by $Int(\mathbb{N})$ the set of intervals with integer endpoints.

Definition 1 (Uncertain Job-Shop Specification).
An uncertain job-shop specification is $\mathcal{J} = (P, M, \prec, \mu, D, U)$ where P is a finite number of tasks, M is a finite set of machines, $\prec$ is a partial-order precedence relation on tasks, $\mu : P \to M$ assigns tasks to machines, $D : P \to Int(\mathbb{N})$ 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 the lower- and upper-bounds of the intervals, respectively. The set $\Pi(p) = \{p' : p' \prec p\}$ denotes all the predecessors of p, namely the tasks that need to terminate before p starts. In the standard job-shop scheduling problem, $\prec$ decomposes into a disjoint union of chains (linear orders) called jobs.

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

Definition 2 (Schedule). *Let $\mathcal{J} = (P, M, \prec, \mu, D, U)$ be an uncertain job-shop specification and let $\mathcal{J}(d)$ be a deterministic instance. A feasible schedule for $\mathcal{J}(d)$ is a function $s : P \to \mathbb{R}_+$, where $s(p)$ defines the start time of task p, satisfying:*
1) Precedence*: For every p, $s(p) \geq \max_{p' \in \Pi(p)}(s(p') + d(p'))$.*
2) Mutual exclusion*: For every p, p' such that $\mu(p) = \mu(p')$*

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

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

The schedule length is the termination time of the last task, i.e. $\max_{p \in P}(s(p) + d(p))$. An *optimal schedule* for $\mathcal{J}(d)$ is a schedule having a minimal length.

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

[3] It can be found in `www-verimag.imag.fr/~maler/Papers/uncertain.ps`

Definition 3 (State of Schedule). *The state of a schedule s at time t is $S = (P^f, P^a, c, P^e)$ such that P^f is a downward-closed subset of $(P, \prec)$ indicating the tasks that have terminated (those satisfying $s(p) + d(p) \leq t$), P^a is a set of active tasks currently being executed (those satisfying $s(p) \leq t \leq s(p) + d(p)$), $c : P^a \to \mathbb{R}_+$ is a function such that $c(p) = t - s(p)$ indicates the time elapsed since the activation of p and P^e is the set of enabled tasks consisting of those whose predecessors are in P^f. The set of all possible states is denoted by $\mathcal{S}$.*

Definition 4 (Scheduling Strategy). *A (state-based) scheduling strategy is a function $\sigma : \mathcal{S} \to P \cup \{\bot\}$ such that for every $S = (P^f, P^a, c, P^e)$, $\sigma(S) = p \in P^e \cup \{\bot\}$ and for every $p' \in P^a$, $\mu(p) \neq \mu(p')$.*

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

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

Definition 5 (Residual Problem). *Let $\mathcal{J} = (P, M, \prec, \mu, D, U)$ and let $S = (P^f, P^a, c, P^e)$ be a state. The residual problem starting from S is $\mathcal{J}_S = (P - P^f, M, \prec', \mu', D', P^a)$ where $\prec'$ and μ' are, respectively, the restrictions of $\prec$ and μ, to $P - P^f$ and D' is constructed from D by letting*

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

Likewise a residual instance d_S is an instance restricted to $P^a \cup P^e$ defined as

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

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 $\mathcal{J}_S(d_S)$. If we take d to be the maximal instance, this is exactly the property of the on-line re-scheduling approach described informally in the previous section.

3 Timed Automata for Scheduling Problems

In this section we model the problem using timed automata based on definitions that can be found in [AM01]. We construct for every task p with $D(p) = [l, u]$ a 3-state timed automaton $\mathcal{A}_D$ (Figure 2-(a)) with a waiting state $\overline{p}$, an active state p where the task is executing and a final state $\underline{p}$. The automaton has one clock which is reset to zero upon entering p ("start") and its value determines when a transition to $\underline{p}$ ("end") is taken. This automaton captures all instances: it can stay in p as long as $c \leq u$ and can leave p as soon as $c \geq l$. It represents the possible behaviors of the task *in isolation*, i.e. ignoring precedence and resource constraints. The transition from $\overline{p}$ to p is triggered by a decision of the scheduler, respecting those constraints, while the time of the transition from p to $\underline{p}$ is determined by the instance. When an instance d is given, all the non-determinism is related to scheduler decisions and the behaviors are captured by the automaton $\mathcal{A}_d$ of Figure 2-(b). The automaton $\mathcal{A}_{D,d}$ of Figure 2-(c) will be used later for computing d-future optimal strategies: it can terminate as soon as $c \geq d$ but can stay in p until $c = u$.

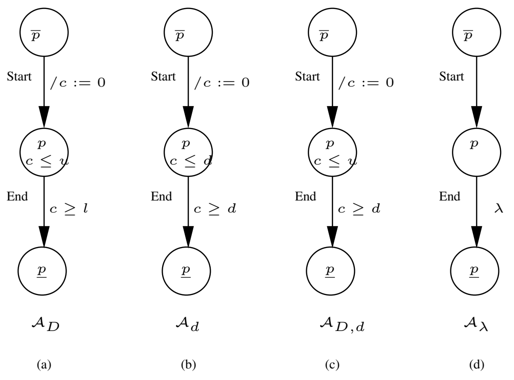

Fig. 2. The generic automaton $\mathcal{A}_D$ for a task p such that $D(p) = [l, u]$. The automaton $\mathcal{A}_d$ for a deterministic instance d. The automaton $\mathcal{A}_{D,d}$ for computing d-future optimal strategies and the automaton $\mathcal{A}_\lambda$ for an exponentially distributed duration. Staying conditions for $\overline{p}$ and $\underline{p}$ are **true** and are omitted from the figure.

The timed automaton for the whole job-shop specification is the composition of the automata for the individual tasks.[4] The composition is rather standard, the only particular feature is the enforcement of precedence and mutual exclusion constraints. This is achieved by forbidding global states in which a task is active before all its predecessors have terminated or in which two or more tasks that use the same resource are active (see [AM01]).

The result of applying this composition to the automata corresponding to the example[5] appears in Figure 3. Since in this example $\prec$ decomposes into two disjoint chains, we can annotate global discrete states with tuples of the form (α^1, α^2) where α^j is either

[4] In the following we will not distinguish between $\mathcal{A}_D$, A_d and $\mathcal{A}_{D,d}$ — the definitions are the same for all of them.

[5] To make things simpler we change J_1 to be completely deterministic, i.e. $J_1 = (m_1, 10) \prec (m_3, 4) \prec (m_4, 5)$.

$\overline{m}$ or m where $m = \mu(p)$ and p is the maximal enabled or active task in the j^{th} chain (or f when the last task in the chain has terminated). For example $(\underline{p}_1, \overline{p}_2, \overline{p}_3, p_4, \overline{p}_5)$ is written as $(\overline{m}_3, m_2)$ and $(\underline{p}_1, \overline{p}_2, \overline{p}_3, \underline{p}_4, \underline{p}_5)$ as $(\overline{m}_3, f)$. For the same reason we can re-use the same clock for all tasks that share the same chain.[6] Note that the automaton is acyclic.

The relation between runs of the automaton and feasible schedules was elaborated in [AM01,A02] where it was shown that solving the (deterministic) job-shop scheduling problem amounts to finding the shortest run (in terms of elapsed time) from the initial to the final state. A configuration of the timed automaton corresponds to a state of the schedule and the residual problem associated with such a state 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 strategy is a function that maps any configuration of the automaton either into one of its transition successors or to the waiting "action". For example, at $(m_1, \overline{m}_3)$ there is a choice between moving to (m_1, m_3) by giving m_3 to J_2 or waiting until J_1 terminates m_1 and letting the environment take the automaton to $(\overline{m}_3, \overline{m}_3)$, from where the conflict concerning m_3 can be resolved in either of the two possible ways.

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

4 Optimal Strategies for Timed Automata

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

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

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

[6] More on the relation between jobs and partially-ordered tasks can be found in [AKM03].

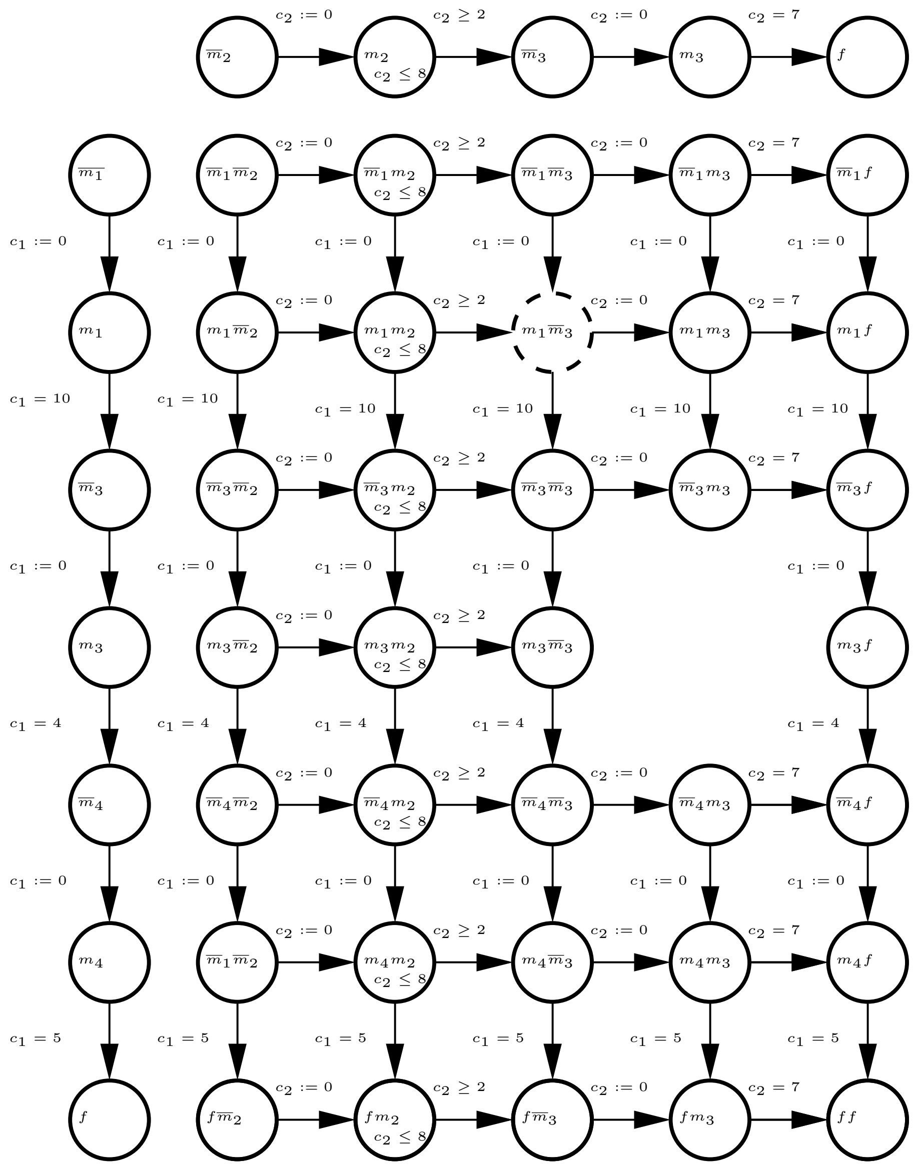

Fig. 3. The global automaton for the job-shop specification. The automata on the left and upper parts of the figure are the partial compositions of the automata for the tasks of J_1 and J_2, respectively. The "hole" at the right stands for the illegal state (m_3, m_3). The dashed state is where a decision of the scheduler is needed.

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

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

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 what will terminate before, m_4 or m_3 and whether the shorter path goes via (m_4,f) or (f,m_3).

$$h(m_4,m_3,c_1,c_2) = \min \left\{ \begin{array}{l} 7 \dot{-} c_2 + h(m_4,f,c_1+7 \dot{-} c_2,\bot), \\ 5 \dot{-} c_1 + h(f,m_3,\bot,c_2+5 \dot{-} x1) \end{array} \right\}$$

$$= \min\{5 \dot{-} c_1, 7 \dot{-} c_2\} = \begin{cases} 5 \dot{-} c_1 \text{ if } c_2 \dot{-} c_1 \geq 2 \\ 7 \dot{-} c_2 \text{ if } c_2 \dot{-} c_1 \leq 2 \end{cases}$$

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 state-space $Q \times V$. In particular, for state $(m_1,\overline{m}_3)$ where we need to decide whether to start m_3 of J_2 or to wait, we obtain:

$$h(m_1,\overline{m}_3,c_1,\bot) = \min\{16, 21 \dot{-} c_1\} = \begin{cases} 16 & \text{if } c_1 \leq 5 \\ 21 \dot{-} c_1 & \text{if } c_1 \geq 5 \end{cases}$$

The extraction of a strategy from h is straightforward: if the optimum of h at $(q,\mathbf{v})$ is obtained via a controlled transition to q' we let $\sigma(q,\mathbf{v}) = q'$ otherwise, when it is obtained via an uncontrolled transition we let $\sigma(q,\mathbf{v}) = \bot$. 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 $c_1 \leq 5$ 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 $c_1 \geq 5$. 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.

The results of [AM01] concerning "non-lazy" schedules imply that there exist an optimal strategy having the additional property that if $\sigma(q,\mathbf{v}) = \bot$ then $\sigma(q,\mathbf{v}') = \bot$ for every $\mathbf{v}' \geq \mathbf{v}$. In other words, if an enabled controlled transition gives the optimum it can be taken as soon as possible. This fact will be used later in the implementation of the strategy.

Existing algorithms for timed automata work on sets, not on functions, and in order to apply them to the computation of h we do the following.[7] Let $\mathcal{A}'$ be an auxiliary automaton obtained from $\mathcal{A}$ by adding a clock T which is never reset to zero. Clearly, if $(q,(\mathbf{v},T))$ is reachable in $\mathcal{A}'$ from the initial state $(s,(\mathbf{0},0))$ then $(q,\mathbf{v})$ is reachable in $\mathcal{A}$ in time T. Let Θ be a positive integer larger then the longest path in the automaton. Starting from $(f,(\bot,\ldots,\bot,\Theta))$ and doing backward reachability we can construct a relational representation of h. More precisely, if $(q,(\mathbf{v},T))$ is backward reachable from $(f,(\bot,\ldots,\bot,\Theta)$in $\mathcal{A}'$ then f is forward reachable in $\mathcal{A}$ from $(q,\mathbf{v})$ within $\Theta - T$ time.

We recall some commonly-used definitions in the verification of timed automata [HNSY94]. A *zone* is a subset of V consisting of points satisfying a conjunction of inequalities of the form $c_i - c_j \geq k$ or $c_i \geq k$. 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 $\{(q,\mathbf{v}) : \mathbf{v} \in Z\}$.

[7] A similar construction was proposed in [NTY00] to implement shortest path algorithm for cyclic timed automata using forward reachability.

Zones and symbolic states are closed under various operations including the following:

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

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

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

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

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

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

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

Algorithm 1 is based on the standard backward reachability algorithm for timed automata. It starts with the final state of $\mathcal{A}'$ 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 1 (Backward Reachability for Timed Automata)

```
Waiting:={$(f, (\bot, \ldots, \bot, \Theta))\}, \emptyset)\}$;
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
R:=Explored;
```

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

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

If q' is a successor via a controlled transition, we move to q', otherwise we wait until a task terminates and an uncontrolled transition is taken. Non-laziness guarantees that we need not revise a decision to wait until the next transition. This concludes our major contribution, an algorithm for computing d-future optimal strategies for the problem of job-shop scheduling under uncertainty.

Theorem 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.*

5 Probabilistic Uncertainty

In this section we sketch the formulation and the solution of the same problem where uncertainty in task durations is considered to be probabilistically distributed. We use exponential distribution and associate with each task a parameter λ such that the time t that the task spends in its active state p satisfies:

$$P(t \geq T) = e^{-\lambda T}.$$

The automaton for a task, depicted in Figure 2-(d), 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 $\overline{p} \xrightarrow{r} \overline{p} \xrightarrow{0} p \xrightarrow{t} p \xrightarrow{0} \underline{p} \xrightarrow{\infty}$, where r is the time chosen by the scheduler to wait before starting p.

A probabilistic version of the example used in the previous section looks like this:

$$J_1 = (m_1, \lambda_1) \prec (m_3, \lambda_2) \prec (m_4, \lambda_3) \qquad J_2 = (m_2, \lambda_4) \prec (m_3, \lambda_5)$$

and it induces a probability distribution on the space of instances, $\mathbb{R}^5_+$. A scheduling strategy is, as before, a mechanism for deciding at every instance whether to start an enabled task or to wait. A strategy together with an instance determines the length of the obtained schedule and our goal is to find a strategy that optimizes the *expected value* (over all instances) of this length.

The automata for the example are similar to those in Figure 3 with λ replacing $[l, u]$. The states of the product automaton admit combinations of controlled and probabilistic transitions. A state like $(\overline{m}_3, \overline{m}_3)$ has two controlled transitions that can be taken immediately. A scheduling strategy will determine which of them should be taken. A state like (m_1, m_2) has two outgoing probabilistic transitions and the instance determines which of them will be taken. However it is possible to compute the expected staying time in the state and the probability of each transition to win the "race". In a state having both types of transitions, such as $(\overline{m}_3, m_2)$, the outcome depends on the strategy. If it decides to wait, the controlled transitions are erased and the evolution depends on the probabilistic race. Otherwise if the strategy chooses a start transition, the rest of the transitions disappear. The important thing is that after determining the strategy the system becomes an ordinary continuous time Markov process with a well-defined expected length for a path from beginning to termination, and our goal is to find a strategy that optimizes this expected length.

The exponential distribution is memoryless, which means that the probability of a transition to be taken does not change with the passage of time.[8] Hence an optimal strategy, like the hole filling strategy of the previous section, depends only on the discrete state and does not need to record clock values.

The optimal strategy, like the future-d-optimal strategies of the previous section, is found by a variant of dynamic programming value iteration. Let $h : Q \to \mathbb{R}_+$ be a function such that $h(q)$ is the best achievable expected value of the 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

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

with parameters $\lambda_1, \ldots, \lambda_k$, leading to states $q_1, \ldots, q_k$, respectively, and l outgoing "start" transitions leading to states $q'_1, \ldots, q'_l$, respectively. A strategy that takes one of the start transitions to a state q'_j spends no time at q and hence the expected time to reach f will be like that of q'_j. On the other hand a strategy that waits might make the environment take any of the "end" transition. Hence

$$h(q) = \min\{h^\perp(q), h(q'_1), \ldots h(q'_l)\}$$

where $h^\perp(q)$ is the expected value of h over all possible outcomes of waiting, computed as:

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

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 be the one taken. These are:

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

The strategy chooses to wait or to take one of the start transitions according to where the minimum is obtained. To the best of our knowledge, this as an unexplored class of continuous-time Markov decision processes for which we can show:

Theorem 2 (Optimal Strategies for Probabilistic Uncertainty). *The problem of finding an optimal strategy for a job-shop specification with exponentially distributed durations is solvable.*

6 Discussion

We have implemented Algorithm 1 using the zone library of Kronos[BDM$^+$98], as well as the hole-filling strategy and the algorithm for the exponential distribution. In our first set of experiments, a d-future optimal strategy based on the worst-case produced schedules that, on the average, are only 2.39% longer than optimal schedules produced by a clairvoyant scheduler. For comparison, the static worst-case strategy deviates from the optimum by an average of 16.18%. The hole-filling strategy based on worst-case prediction achieves good performance (3.73% longer than the optimum). On the other hand, if these strategies are based on nominal instances other than the worst-case, the results are poor, sometimes even worse than a static schedule. So one may conclude that *adaptive pessimism* is a reasonable strategy for this class of problems.

The question of scaling-up the results to larger problems remains open. Currently we can compute d-future optimal strategies for problems with up to 4 jobs, each with 6 tasks. The computation of the strategy for exponential distribution is faster (no clocks and zones) but it is subject to the same type of state explosion. For the deterministic case, we have shown in [AM01] that rather large problems can be solved using forward reachability algorithms that do not use zones (only points in the clock space) and that can use intelligent search strategies to prune the search space (see also [BFH$^+$01]). This is not the case for uncertain problems where backward computations on zones seem unavoidable: Under uncertainty the environment can lead the automaton to a large portion of the discrete state-space and to uncountably-many clock valuations, on which the strategy should be defined. The sub-optimal hole-filling strategy produces good results with much more modest computation by solving a deterministic problem. More details concerning the experimental results and the computational difficulty appear in the expanded version of the paper along with some suggestions for future work.

Acknowledgment. We thank Marius Bozga for his help in the implementation. Comments by Stavros Tripakis, Ed Brinksma and Albert Benveniste improved the presentation.

References

[A02] Y. Abdedaïm, *Scheduling with Timed Automata*, PhD Thesis, INPG, 2002.

[AKM03] Y. Abdedaïm, A. Kerbaa and O. Maler Task Graph Scheduling using Timed Automata, *Proc. FMPPTA'03*, to appear, 2003.

[AM01] Y. Abdedaïm and O. Maler, Job-Shop Schedusling using Timed Automata in *Proc. CAV'01*, 478–492, LNCS 2102, Springer 2001.

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

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

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

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

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

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

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

[NY00] P. Niebert and S. Yovine, Computing Optimal Operation Schemes for Chemical Plants in Multi-batch Mode, *Proc. HSCC'00*, 338–351, LNCS 1790, Springer, 2000.