ELSEVIER

Annual Reviews in Control 31 (2007) 1–15

Annual Reviews in Control

www.elsevier.com/locate/arcontrol

# On optimal and reasonable control in the presence of adversaries☆

Oded Maler

*CNRS-Verimag, 2, av. de Vignate, 38610 Gières, France*

Received 31 March 2005; accepted 24 February 2007

**Abstract**

This paper constitutes a sketch of a unified framework for posing and solving problems of optimal control in the presence of uncontrolled disturbances. After laying down the general framework we look closely at a concrete instance where the controller is a scheduler and the disturbances are related to uncertainties in task durations.


*Keywords:* Discrete event systems; Hybrid systems

## 1. Introduction

The design of mechanisms for achieving certain goals in complex and not fully-predictable environments is a universal human activity, often aided by mathematical models. The choice of the class of models is influenced, of course, by their adequacy for the application domain and the existence of useful analysis techniques, but it may also depend on historical, cultural, sociological and accidental factors that result in a particular distribution of models, techniques and terminologies over the space of applications and scientific communities.

The present paper attempts to distill the *essence* of many of the models and techniques employed for this purpose by different disciplines and communities under diverse titles such as System Verification, Controller Synthesis, Sequential Decision Making, Game Theory, Markov Decision Processes, Planning in AI and Robotics, Optimal and Model Predictive Control, Shortest Path Algorithms, Dynamic Programming, Optimization, Reinforcement Learning, Differential Games and more. I hope to convince the reader that in all these contexts, the basic problem amounts to finding an optimal/winning strategy in some generic two-player dynamic games, where the optimality is defined according to some domain-specific cost function, and a particular way to quantify over the behaviors of the ''adversary''. Moreover, I intend to show that there are basically three generic methods used to solve this problem: bounded-horizon optimization, dynamic programming and forward search.

Each of the disciplines in which these issues are treated has its own vocabulary. For example, what is called in one context a *strategy*, can be called elsewhere a *policy*, a *feed-back law*, a *controller*, a *reactive plan* or a *scheduler*. Even worse, the same term can mean different things to different audiences. I did my best to pick each time the term (or terms) that I felt are the most appropriate for the discussion and tried not to switch between terms too often. In any case I apologize for the potential inconvenience for those who are accustomed to their own native language. I also tried not to bias the description toward my own automata-theoretic background.

Although this paper is free of any ''new original results'' and does not say anything that is not known in some of the many disciplines and communities that occupy themselves with these issues, I think that putting all this together in the present form has some ''synergetic'' value that goes beyond a collection of informal definitions, results and algorithms. Moreover, such a unified framework may eventually facilitate a more concentrated effort toward meeting the computational challenges posed by controller synthesis.

The paper has two main parts. In the first I lay down a kind of a ''special theory of everything'' where system design is viewed as synthesizing an optimal strategy in some dynamic game. I start in Section 2 with a general discussion on how to *define* the performance of a system which is subject to external disturbances. Section 3 introduces the generic model that we use, a dynamic ''multi-stage'' game between a controller and its

☆ This research was supported in part by the European Community projects IST-2001-33520 CC (Control and Computation) and IST-2001-35304 AMETIST (Advanced Methods for Timed Systems).
*URL:* www-verimag.imag.fr/~maler/.

1367-5788/$ – see front matter 
doi:10.1016/j.arcontrol.2007.02.001

environment and discusses various ways to assign costs to behaviors and define the optimal control problem. The restriction of the problem to behaviors of bounded length is the topic of Section 4 where it is reduced (for discrete-time systems) to standard finite-dimensional constrained optimization. This reduction is rather straightforward in the absence of an adversary but is less so in its presence, which calls for a more subtle notion of optimality, achieved by a class of techniques known as *dynamic programming*, described in Section 5. While dynamic programming provides, in principle, a complete solution to the problem, it does not scale up as the number of state variables grows and, as described in Section 6, one often has to resort to the alternative method of heuristic forward *search*. Section 7 summarizes the first part and discusses the surprising lack of explicit modeling of adversaries in certain popular control models.

In the second part of the paper I focus on one concrete instance of this scheme, the problem of scheduling under bounded uncertainty which is modeled as a discrete game on continuous time using the *timed automaton* model. Section 8 attempts to convince the reader that scheduling falls into the class of problems described in the first part. The job shop problem is presented in Section 9 along with its traditional solution scheme based on non-convex "combinatorial" optimization. In Section 10 we show how this problem can be solved using shortest path algorithms on timed automata. Section 11 extends the problem by considering bounded uncertainty in task durations, while Section 12 sketches a dynamic programming algorithm that can find adaptive scheduling policies that perform better than static schedules tailored for the worst scenario. Some thoughts on modeling, analysis and research in general conclude the paper.

## 2. Static optimization

Throughout this paper we will be concerned with situations that resemble two-player *games*. One player, henceforth the *controller*, represents the system that we want to design while the other player, the *environment*, represents external disturbances beyond our control. The controller chooses actions $u \in U$, the environment picks $v \in V$ and these choices determine the outcome of the game. The controller wants the outcome to be as good as possible, according to some pre-defined criterion, while the environment, unless one is paranoid, is indifferent to the results. It should be noted that unlike the classical setting of game theory, where *both* players are supposed to have a utility function and questions are often related to the outcome of games where each of them tries to optimize its own objective function, we do not make assumptions concerning the goals of the adversary. The adversary enters the picture because we take *all* its potential actions into consideration while evaluating and comparing the potential actions of the controller.[1]

Taking the adversary into account leads to a departure from standard optimization of functions because we optimize something that depends not only on our own actions. Let us illustrate this point using a simple one-shot game of the type introduced in the seminal work of von Neumann and Morgenstern.

Let $U = \{u_1, u_2\}$, $V = \{v_1, v_2\}$ and let the outcome be defined as a function $c : U \times V \to \mathbb{R}$ which can be given as a table

| $c$ | $v_1$ | $v_2$ |
|---|---|---|
| $u_1$ | $c_{11}$ | $c_{12}$ |
| $u_2$ | $c_{21}$ | $c_{22}$ |

We want to choose among $u_1$ and $u_2$ the one that minimizes $c$ but since different choices of $v$ may lead to different values, we need to specify how to take these values into account while evaluating a specific choice of $u$. There are basically three generic approaches for evaluating our choices:

- *Worst case*: each action of the controller is evaluated according to the worst outcome that may result from taking the action. Hence we choose the $u$ for which the maximal $v$-induced cost is minimal:

  $$u = \operatorname{argmin} \max \{c(u, v_1), c(u, v_2)\}.$$

- *Average case*: the environment is modeled as a stochastic agent acting randomly according to a probability function $p : V \to [0, 1]$ and the controller actions are evaluated according to the *expected value* of $c$. We choose the $u$ which minimizes the average of the $v$-induced costs:

  $$u = \operatorname{argmin}\ p(v_1) \cdot c(u, v_1) + p(v_2) \cdot c(u, v_2).$$

- *Typical case*: the evaluation is done with respect to a *fixed* element of $V$, say $v_1$, which represents the most "typical" behavior of the adversary. This amounts to ignoring the existence of uncontrolled disturbances and the problem is reduced to ordinary optimization:

  $$u = \operatorname{argmin} c(u, v_1).$$

Let us note that when $c$ is a continuous function over continuous domains $U$ and $V$, finding the optimum cannot be done by exploring a finite table but rather by analytic or numerical methods. For such functions, average case analysis stays within the standard optimization framework, that is, optimization of a well-behaving *real-valued* function, while the worst-case min–max analysis does not. This may partially explain why in domains such as continuous control stochastic modeling of disturbances is much more popular.

## 3. Dynamic games

### 3.1. Dynamics

A dynamic game is a game where the players are engaged in an *ongoing interaction* extended over time. In the computer science context, the term *reactive systems*, coined by Harel and Pnueli, is used to denote such objects and distinguish them from traditional "transformational" programs that read their input at the beginning of their execution and do not interact with the external world until they produce their output upon termination.

[1] There is a partial overlap between this approach and what is sometimes called *games against nature*.

A game is characterized by a state-space $X$, action domains $U$ and $V$ for the two players and a dynamic rule of the form

$$x' = f(x, u, v)$$

stating that at each time instant the "next" value of $x$ (denoted by $x'$) is a function of its current value and of the actions of both players. In this part of the paper we focus on *discrete-time "synchronous" games* where the time index is often assumed to be uniform and both players take actions simultaneously at every step. In this case the dynamics can be written as a recurrence equation of the form

$$x_t = f(x_{t-1}, u_t, v_t).$$

But let us keep in mind the existence of other models such as *differential games* on continuous time defined via

$$\dot{x} = f(x, u, v)$$

or games with a more "asynchronous" flavor where actions may occur at non-periodic time instants (event-triggered rather then time-triggered) and where actions of both players need not occur simultaneously. Such asynchronous games will be used in the second part of the paper to model scheduling problems.

We assume all games to start from an initial state $x_0$ and use the notation $\bar{x} = x[0], x[1], \ldots, x[k]$ for sequences of states that we call *behaviors* (they are also known as *runs* in the discrete context and *trajectories* in the continuous). Likewise, we will use $\bar{u}$ and $\bar{v}$ for sequences of players actions. A pair of action sequences $\bar{u}$ and $\bar{v}$ issued by the two players, respectively, induces a unique behavior $\bar{x}$ of the system. We use the predicate (constraint) $B(\bar{x}, \bar{u}, \bar{v})$ to denote the fact that $\bar{x}$ is induced by $\bar{u}$ and $\bar{v}$. Formally, it is defined as:

$$B(\bar{x}, \bar{u}, \bar{v}) \quad \text{iff} \quad x[0] = x_0, \quad x[t] = f(x[t-1], u[t], v[t]) \quad \forall\, t$$

It is sometimes useful to view the game as a *labeled directed graph* whose nodes are the elements of $X$ (states) and its edges (transitions) are all the pairs $(x, x')$ such that $f(x, u, v) = x'$ for some $u$ and $v$. When $X$, $U$, and $V$ are discrete, one can visualize the game as in Fig. 1. In the same spirit we can write $B(\bar{x}, \bar{u}, \bar{v})$as:

$$x[0] \rightarrow^{u[1],v[1]} x[1] \cdots \rightarrow^{u[k],v[k]} x[k].$$

It goes without saying that when the state space and actions range over continuous domains, the game cannot be drawn as a discrete graph nor written as a transition table, but rather be defined using some traditional mathematical description, for example a linear game of the form

$$x' = Ax + Bu + Cv$$

but, nevertheless, the reader is encouraged to stretch the imagination and try to "see" the underlying graph structure.

### 3.2. Costs

After having specified which behaviors can be induced by actions of the two players, we need to assign performance measures (costs) to such behaviors in order to compare them and prefer some over the others. The first step is to assign costs to *individual* behaviors, and then evaluate each strategy according to some approach for combining the costs of all adversary-induced behaviors it will generate, as discussed in Section 2.

| $x$ | $u$ | $v$ | $x'$ |
|---|---|---|---|
| $x_0$ | $u_1$ | $v_1$ | $x_0$ |
| | | $v_2$ | $x_1$ |
| | $u_2$ | $v_1$ | $x_1$ |
| | | $v_2$ | $x_3$ |
| $x_1$ | $u_1$ | $v_1$ | $x_2$ |

. . . . . .

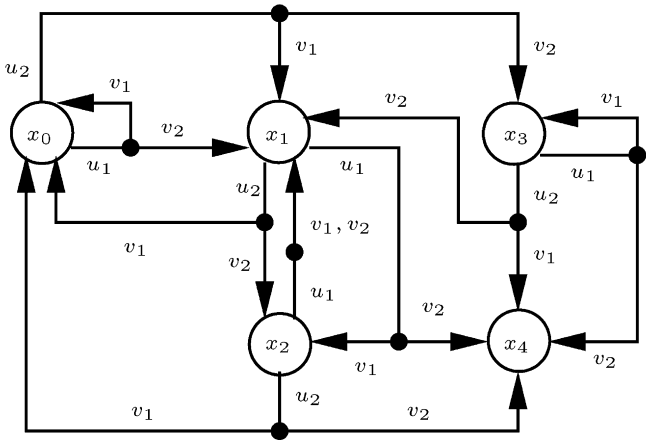


Fig. 1. A dynamic game in a tabular and graphical description. The $u$-labels appear before the $v$-labels just for convenience.

A common way to assign costs to behaviors is to start by associating *local* costs to every transition with $c(x, u, v)$ reflecting the cost of the actions $u$ and $v$ taken at $x$ and the goodness of the state $x' = f(x, u, v)$ reached after the transition. In the sequel we sometimes restrict ourselves to cost functions that depend only on $x$ and $u$ or to purely state-based cost functions of the form $c : X \rightarrow \mathbb{R}$.

The nature of cost functions depends on the application domain. In discrete verification $c$ is typically (and implicitly) a $\{0, 1\}$-valued function such that $c(x) = 1$ refers to "bad" states that we want to avoid (or eventually leave) and $c(x) = 0$ characterizes "good" states that we want to stay in forever (or eventually reach). In continuous domains, $c(x)$ may indicate some norm or distance from a reference state. Sometimes the choice of the cost function is influenced less by its adequacy for the problem and more by the existence of a corresponding optimization method, especially when the optimum is to be computed analytically.

The next step is to "lift" the local cost function defined over states or transitions to a cost function on sequences. In discrete systems one can do it by letting

$$\bar{c}(\bar{x}) = \max \{c(x[t]) : t = 1, \ldots, k\}$$

so that $c(\bar{x}) = 1$ iff a bad state is encountered, or by

$$\bar{c}(\bar{x}) = \min\{c(x[t]) : t = 1, \ldots, k\}$$

so that $c(\bar{x}) = 0$ iff a good state has been reached. The most natural and popular way to assign costs to behaviors is summation of local costs

$$\bar{c}(\bar{x}) = \sum_{t=1}^{k} c(x[t]),$$

which for continuous-time systems can be replaced by integration and called fancy names such as norms in function spaces.

Note that for systems admitting a set $F$ of target states such that $c(x) = 0$ when $x \in F$ and $c(x) > 0$ otherwise, summation expresses the time or cost to reach the target, hence the tight relationship between optimal control and shortest paths problems on graphs. To avoid divergence of the cost function with the length of the behavior, summation is often combined with *discounting*

$$c(\bar{x}) = \sum_{t=1}^{k} 2^{-t} c(x[t]),$$

with averaging or with restriction to instances taken from a shifting time window. All these approaches can be extended naturally to incorporate the cost of actions and define $\bar{c}(\bar{x}, \bar{u})$ or $\bar{c}(\bar{x}, \bar{u}, \bar{v})$.

### 3.3. Sub models

The game model can be reduced, by suppressing either one of the players, or both, into sub models (see Fig. 2) which are central to various disciplines. When the controller has no choice, either by letting $U$ be a singleton or equivalently using a dynamical model of the form $x' = f(x, v)$, the problem is transformed into *evaluating* the performance of a *given* controller amidst disturbances. This is the subject matter of discrete formal verification, where the adversary is sometimes disguised as a non-deterministic dynamics $g : X \to 2^X$ which can be retrieved as

$$g(x) = \bigcup_{v \in V} f(x, v).$$

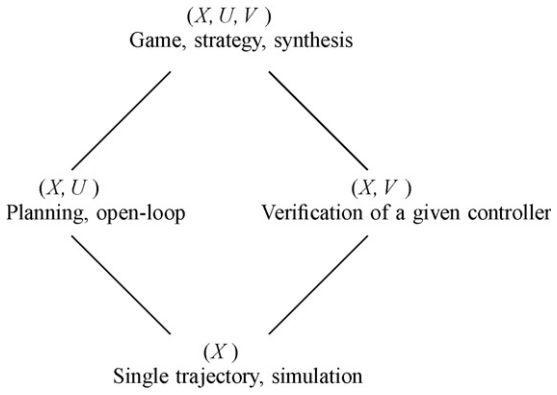


Fig. 2. Sub models obtained by suppressing one or two of the players.

The questions one typically asks concerning such models can be formulated as: what is the worst scenario that the external environment can induce in the system. Let us note that such systems can be obtained from the general model after the controller has been synthesized, and the comparison between candidate controllers reduces to a comparison between systems of the $f(x, v)$ type.

A dual model is obtained when the adversary is suppressed and the dynamics considered is of the form $x' = f(x, u)$. In this case the problem of finding an optimal strategy is reduced to finding an optimal sequence. Such approaches are common in robotics path planning and AI planning in general. When both players are suppressed we obtain a deterministic dynamical system of the form $x' = f(x)$ in which there is only one trajectory emanating from $x_0$. In the sequel, when we describe different solution methods, we will first illustrate each of them on the adversary-free model $x' = f(x, u)$ and then extend them for the full model.

## 4. Bounded horizon problems

We will now restrict ourselves to situations where we compare only behaviors of a fixed finite length. There are several reasons to focus on bounded decision horizons. The first is that there are certain problems of the "control to target" or "shortest path" type, where all reasonable behaviors converge to a goal state in a bounded number of steps (but via paths of different costs). Another reason is the common sense intuition that as we look further into the future, our models become less reliable, and hence it is better to plan for a shorter horizon and revise the plan during execution (this is the basis of model-predictive control). Finally, bounded horizon problems in discrete time can be reduced to *finite dimensional* optimization problems.

We first illustrate the formulation of the problem for *adversary-free* situations with dynamics of the form $x' = f(x, u)$. In this case we look for a sequence $\bar{u} = u[1], \ldots, u[k]$ which is the solution of the constrained optimization problem

$$\min_{\bar{u}} \; c(\bar{x}, \bar{u}) \;\text{ subject to } B(\bar{x}, \bar{u}).$$

The fact that $\bar{x}$ is the result of following the dynamics $f$ under control $\bar{u}$ is part of the *constraints* of the problem. For linear dynamics, specified by $x' = Ax + Bu$, a local linear cost function, say $c(x, u) = ax + bu$, and summation, the problem reduces to standard linear programming:

$$\min_{\bar{u}} \sum_{t=1}^{k} ax[t] + bu[t] \;\text{ subject to } \bigwedge_{t=2}^{k} x[t] = Ax[t-1] + Bu[t].$$

In discrete verification one is often concerned with the dual question: find the worst behavior that the adversary may induce. There, since the dynamics and cost are defined logically, the problem reduces to Boolean satisfiability, that is, is there a behavior $\bar{x}$ satisfying the dynamic constraints such that $\bar{c}(\bar{x}) = 1$. This problem is known as *bounded model checking* with the adjective "bounded" used to distinguish it from the

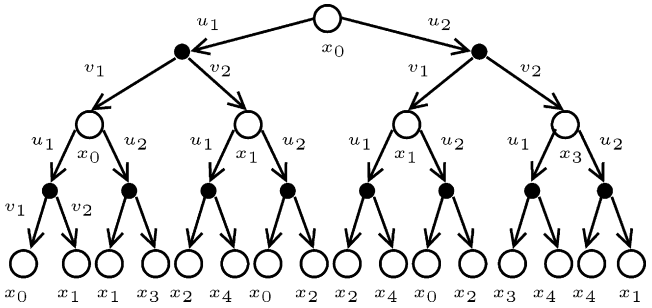

Fig. 3. A game tree of depth 2 obtained by unfolding the game graph of Fig. 1.

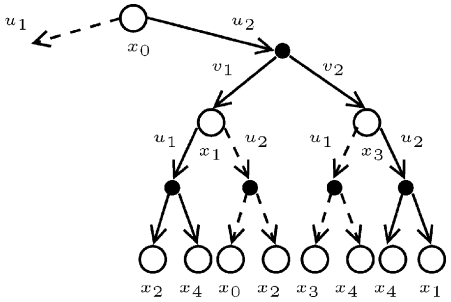

Fig. 4. It might be better to apply $u_1$ after the adversary has chosen $v_1$ and apply $u_2$ otherwise.

traditional methods of algorithmic verification which are closer in spirit to dynamic programming described in the next section.

If we have at our disposal a constrained optimization procedure for the domain in question, we can compute the desired $\bar{u}$ and solve the problem. It is important to note that in the absence of external disturbances, the generated behavior $\bar{x}$ is *completely determined* by $\bar{u}$ and no feed-back from $x$ is needed. The control "strategy" reduces to an open-loop[2] *plan*: at each time instant $t$ apply the element $u[t]$ of $\bar{u}$. Such a plan could be rephrased as a feed-back function (strategy) $s$ defined over all $x[t]$ in $\bar{x}$ as $s(x[t]) = u[t+1]$ but this would be an overkill.

Let us now return to the full model with an adversary and use, without loss of generality, the worst-case criterion. The optimal control sequence $\bar{u}$ is the solution of

$$\min_{\bar{u}} \ \max_{\bar{v}} \ \bar{c}(\bar{x}, \bar{u}) \ \text{ subject to } B(\bar{x}, \bar{u}, \bar{v}). \tag{1}$$

As an illustration, consider the game of Fig. 1 and a time horizon $k = 2$. The structure of the $B$ relation corresponds to the game tree of Fig. 3. We can enumerate all the 4 possible control sequences and write the cost they induce (letting $c(x_i) = c_i$) as:

$$\begin{aligned}
u_1u_1 &: \max\{c_0 + c_0, c_0 + c_1, c_1 + c_2, c_1 + c_4\}\\
u_1u_2 &: \max\{c_0 + c_1, c_0 + c_3, c_1 + c_2, c_1 + c_0\}\\
u_2u_1 &: \max\{c_1 + c_2, c_1 + c_4, c_3 + c_3, c_3 + c_4\}\\
u_2u_2 &: \max\{c_1 + c_0, c_1 + c_2, c_3 + c_4, c_3 + c_1\}
\end{aligned}$$

The sequence which minimizes these values is the optimal open-loop control that can be achieved. For discrete systems, where min and max coincide with $\exists$ and $\forall$, finding the optimal sequences reduces to a simple version of satisfiability of quantified Boolean formulae (QBF).

Using feed-back, however, one can do better. While the choice of $u[1]$ is done without any knowledge of the adversary's action, the choice of $u[2]$ is done *after* the effect of $v[1]$, that is, the value of $x[1]$, is known. Consider the case where $u[1] = u_2$ and we need to choose $u[2]$. If, for example, $\max\{c_2, c_4\} < \max\{c_0, c_2\}$ but $\max\{c_3, c_4\} > \max\{c_1, c_4\}$ then the optimal thing to do would be to apply $u_1$ when $x[1] = x_1$ and $u_2$ when $x[1] = x_3$ (Fig. 4).

A *control strategy* is thus a function $s : X \to U$ telling the controller what to do at any reachable state of the game. In order to formulate the problem of finding an optimal strategy as a constrained optimization problem we need first to replace $B(\bar{x}, \bar{u}, \bar{v})$ by a predicate indicating the fact that $\bar{x}$ is the behavior of the system in the presence of disturbance $\bar{v}$ when the controller *employs strategy* $s$:

$$\begin{aligned}
B(\bar{x}, s, \bar{v}) \quad \text{iff } & x[0] = x_0,\\
& u[t] = s(x[t-1]) \ \forall t,\\
& x[t] = f(x[t-1], u[t], v[t]) \ \forall t
\end{aligned}$$

Finding the best (worst-case) strategy $s$ then becomes the following "second-order" optimization problem:

$$\min_{s} \ \max_{\bar{v}} \ \bar{c}(\bar{x}, \bar{u}, \bar{v}) \ \text{ subject to } B(\bar{x}, s, \bar{v}). \tag{2}$$

Finding an optimal strategy, at least when done naively, is usually much more difficult than finding an optimal sequence. For discrete finite-state systems there are $|U|^{|X|}$ potential strategies and each of them induces $|V|^k$ behaviors of length $k$.

As mentioned in Section 3.3, strategy synthesis can be viewed as removing all but one possible choices for $u$, thus reducing the game model into a model where only the adversary has a choice. The result of applying this procedure to the game of Fig. 1 is shown in Fig. 5. The resulting set of all possible behaviors is a $V$-labeled tree of Fig. 6.

Before moving to dynamic programming let us note some obvious shortcomings of worst-case analysis. Suppose we are in a state where the adversary can either drive us to an extremely bad state $x_1$ (say, the end of the world) or to a state $x_2$ when we have to choose between a good action $u_1$ and a bad action $u_2$. The definition of optimality in (2) cannot distinguish between a strategy $s$ with $s(x_2) = u_1$ and a strategy $s'$ with $s'(x_2) = u_2$

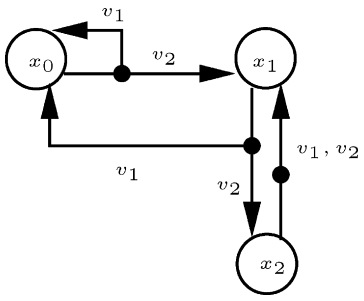

Fig. 5. The system $x' = f(x, v)$ obtained from the game of Fig. 1 by a strategy $s$ satisfying $s(x_0) = u_1$, $s(x_1) = u_2$ and $s(x_2) = u_1$. Note that since $x_3$ and $x_4$ are not reachable from $x_0$ under this strategy, the value of $s$ need not be specified for them.

[2] This is the essential difference between instructions in the style "cook for 5 min" and "cook until the water boils".

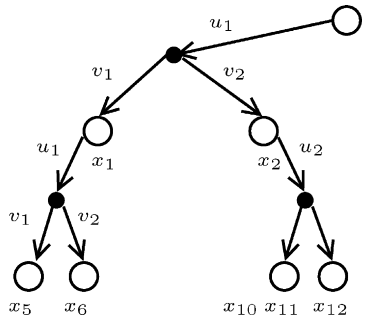


Fig. 6. The set of (adversary-induced) behaviors of length 2 for the system of Fig. 5.

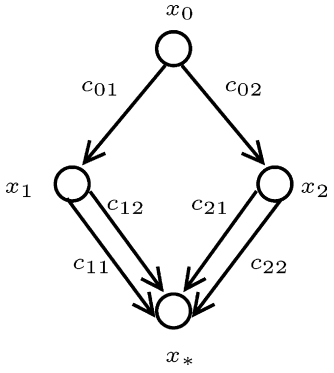


Fig. 7. A shortest path problem without adversary.

because anyway, the maximal cost will be obtained at $x_1$. This is counter-intuitive and we would like to have a criterion that will prefer $s$ over $s'$. This criterion, which is cumbersome to define using constraints, requires a strategy to be optimal not only for the original game starting from $x_0$, but also for all the "residual" games starting from every other state. For this example, although $s$ and $s'$ obtain the same worst-case performance from $x_0$, $s$ is better than $s'$ from $x_2$ and will be preferred. Of course, if an end-of-the-world transition is considered possible from all states, the worst-case criterion fails to be useful.

## 5. Dynamic programming

Dynamic programming, or backward value iteration, is a technique, advocated by Bellman, for computing optimal strategies in an incremental way. We illustrate dynamic programming first using the following adversary-free shortest path problem. Let $x_* \in X$ be a designated target state, from which all actions induce self-loops with zero cost. All other transitions have local costs $c(x_i, u_j) = c_{ij}$. The cost of a path

$$x[0] \overset{u[1]}{\rightarrow} x[1] \cdots \overset{u[k]}{\rightarrow} x[k]$$

is infinite if $x[k] \neq x_*$ and

$$\bar{c}(\bar{x}, \bar{u}) = \sum_{t=2}^{k} c(x[t-1], u[t]),$$

otherwise. Our goal is to find the minimal-cost sequence.

Dynamic programming uses an auxiliary function, known as *value function* or *cost-to-go*, $\vec{\mathcal{V}} : X \to \mathbb{R}$ such that $\vec{\mathcal{V}}(x)$ is the performance of the optimal strategy for the sub-game starting from $x$. In our adversary-free example, this is the length of the shortest path from $x$ to $x_*$. Consider the graph of Fig. 7 which has an additional simplified structure: it is acyclic and all paths that reach a state $x$ from $x_0$ have the same number of transitions, hence its state space can be partitioned into levels according to the number of transitions from $x_0$. The computation of the value function for such graphs is very simple and corresponds to propagation from $x_*$ backwards, starting with:

$$\vec{\mathcal{V}}(x_*) = 0,$$
$$\vec{\mathcal{V}}(x_1) = \min\{c_{11}, c_{12}\},$$
$$\vec{\mathcal{V}}(x_2) = \min\{c_{21}, c_{22}\},$$

and $\vec{\mathcal{V}}(x_0)$ is computed as

$$\begin{aligned}&\min\{c_{01} + \vec{\mathcal{V}}(x_1), c_{02} + \vec{\mathcal{V}}(x_2)\} = \\ &\min\{c_{01} + \min\{c_{11}, c_{12}\}, c_{02} + \min\{c_{21}, c_{22}\}\} = \\ &\min\{c_{01} + c_{11}, c_{01} + c_{12}, c_{02} + c_{21}, c_{02} + c_{22}\}\end{aligned}$$

The last equality is an instance of the Bellman–Dijkstra principle which can be formulated as follows. If $\bar{x}$ is the optimal path from $x_0$ to $x_*$, then for any factorization of $\bar{x}$ into a prefix $\bar{x}_1$ and a suffix $\bar{x}_2$ such that $x_0 \to^{\bar{x}_1} x \to^{\bar{x}_2} x_*$, $\bar{x}_1$ is the optimal path from $x_0$ to $x$ and $\bar{x}_2$ is the optimal path from $x$ to $x_*$. This fact can be expressed, for acyclic graphs, as:

$$\vec{\mathcal{V}}(x) = \min_u (c(x,u) + \vec{\mathcal{V}}(f(x,u)))$$

and in the more general case of graphs admitting cycles as:

$$\vec{\mathcal{V}}(x) = \min\{\vec{\mathcal{V}}(x), \quad \min_u (c(x,u) + \vec{\mathcal{V}}(f(x,u)))\} \tag{3}$$

Eq. (3) is a fixed-point equation on $\vec{\mathcal{V}}$ of the form

$$\vec{\mathcal{V}} = F(\vec{\mathcal{V}}) = \min(\vec{\mathcal{V}}, \ell(\vec{\mathcal{V}}))$$

with $\ell(\vec{\mathcal{V}})$ being a "local improvement" operator on value functions which, as long as $\vec{\mathcal{V}}$ is not optimal, may suggest a shorter path from $x$ to $x_*$ for at least one $x$. As is the case with many equations of this type, the solution is the limit of the sequence of functions $\vec{\mathcal{V}}_0, \vec{\mathcal{V}}_1, \ldots$ generated by the following recurrence relation:

$$\vec{\mathcal{V}}_0(x) = \begin{cases} 0 & \text{when } x = x_* \\ \infty & \text{otherwise} \end{cases}$$
$$\vec{\mathcal{V}}_{i+1}(x) = \min\{\vec{\mathcal{V}}_i(x), \quad \min_u (c(x,u) + \vec{\mathcal{V}}(f(x,u)))\}$$

The termination of this iteration, that is, reaching the fixed point where no further improvement is possible, in a finite number of steps, is guaranteed in many settings, for example when all costs are positive.

A nice feature of this *value iteration* scheme is that it can be adapted very easily to adversarial situations, as well as to other cost functions and other ways to integrate over adversary actions. Consider an adversarial version of the shortest path problem, where the goal is to find a strategy for which the worst-case shortest path (among the adversary-induced paths) is optimal. The value function can be computed by replacing the

local improvement operator of (3) by

$$\min_u \max_v (c(x,u,v) + \vec{\mathcal{V}}(f(x,u,v))). \quad (4)$$

If we are interested by the *average-case* cheapest path, we fall into the realm of the so-called Markov decision processes whose local improvement operator is defined as

$$\min_u \sum_v p(x,v) \cdot (c(x,u,v) + \vec{\mathcal{V}}(f(x,u,v))).$$

Verification for safety properties by backward reachability where $\vec{\mathcal{V}}(x) = 1$ for all bad states and states that may lead to them is done using the operator

$$\max_v (\max\{c(x), \vec{\mathcal{V}}(f(x,v))\}).$$

Controller synthesis for safety (also known as supervisory control for discrete-event systems) uses the operator

$$\min_u \max_v (\max\{c(x), \vec{\mathcal{V}}(f(x,u,v))\})$$

In the case of verification, $\vec{\mathcal{V}}_i(x) = 0$ if all paths of length not greater than $i$ do not reach the bad set from $x$, and in the case synthesis, $\vec{\mathcal{V}}_i(x) = 0$ if there is a controller that can avoid reaching bad states from $x$ for at least $i$ steps. Hence when the fixed point is reached, $\vec{\mathcal{V}}_i(x) = 0$ for states which are safe forever (or can be made so by the controller).

To summarize, this elegant procedure is guaranteed (if it converges) to find the optimal value of the game from any state, including $x_0$. It is then straightforward to extract the optimal strategy from the value function: just take for each $x$ the $u$ that achieves the local optimum. For finite-state systems with positive transition costs, finite convergence is guaranteed in time polynomial in the size of the transition graph, which is better than the exponential enumeration of strategies. However, this is not of much comfort in many situations where the transition graph itself is *exponential* in the number of system variables.

In continuous domains (on continuous time), where the enumeration of all strategies is not an option, the value function is the solution of a partial differential equation known as the Hamilton–Jacobi–Bellman–Isaacs equation, typically solved by discretization of space and time. A notable exception is the case of linear systems with a quadratic cost function where the value function and the controller can be computed analytically.

The major drawback of dynamic programming is the need to compute $\vec{\mathcal{V}}$ for too many states, some not reachable from $x_0$ at all, and some not reachable by any reasonable strategy. This *state-explosion* problem, also known as the *curse of dimensionality*, prevents the straightforward application of the algorithm to systems having a large state-space.

## 6. Forward search

Dynamic programming and optimal control as described in the previous section, provides a very pleasant framework for the mathematically inclined. Here you have a canonical object, the optimal value function, to which you are guaranteed to converge, either within a finite number of steps (in the discrete case) or in some traditional sense of applied mathematics (in the continuous case). Many nice theorems can be proved on variants of the problem. However, for practical purposes, all this scheme breaks down for computational complexity reasons as soon as the number of state variables goes beyond a dozen or so, and often much earlier than that. Consequently if we want to come out with a reasonable strategy, optimal or not, we must give up the exhaustive exploration of all parts of the state space.

Similar problems have been encountered by researchers, mostly in what is known as AI, who attempted to develop good strategies for playing Chess and other games of combinatorial nature admitting prohibitively-large state spaces. Other variants of this problem can be found in Robotics when one has to find paths amidst obstacles. The idea is rather simple and common sense: instead of exploring all possible sequences (in the adversary-free case) or strategies, explore only a reasonably-small subset of them and pick the optimal among those. The major question is, of course, how to direct the search toward the more interesting parts of the state space using domain-specific heuristics.

In the rest of this section we focus on search methods that are *aligned with the progress of time*, that is, they construct partial paths or partial strategies from $x_0$ onward, determining $u[t]$ before $u[t+1]$. It is worth mentioning, however, that methods based on partial exploration of arbitrary solution spaces are common in all areas of optimization and can be invoked to solve bounded-horizon formulations of the controller synthesis problem like those described in Section 4. Such methods need not necessarily work in a ''chronological'' order.

Let us look again at the shortest-path problem without adversary, and assume we deal with finite acyclic graphs. The fixed-point characterization of the value function (3) can be interpreted as the following recursive algorithm for computing $\vec{\mathcal{V}}$ for a given state:

**Algorithm 1** (*Recursive computation of* $\vec{\mathcal{V}}$).

```
real proc V(x)

if x = x*
   return(0)
else
   Val := ∞
   forall u ∈ U
      Val' := c(x,u) + V(f(x,u))
      Val := min{Val, Val'}
   return(Val)
```

When this procedure is called with $x_0$ as an argument and runs to completion, it explores all the paths of the system in a *depth-first* manner and collects the costs as it returns from the recursion. This procedure is redundant in the sense that it recomputes the value of $x$ each time it is reached via another path. This potentially-exponential blow up can be avoided using the Bellman–Dijkstra principle by memorizing $\vec{\mathcal{V}}(x)$ upon returning from the recursion and using this value when $x$ is

subsequently encountered. The procedure maintains an array $\vec{\mathcal{V}}$ initialized to infinity in which the value function of a state is stored once computed.

**Algorithm 2** (*Recursion and memorization*).

```
real proc Value(x)

if x = x_*
    return(0)
else
    if V⃗(x) = ∞
        forall u ∈ U
            Val' := c(x,u) + Value(f(x,u))
            V⃗(x) := min{V⃗(x), Val'}
    return(V⃗(x))
```

These algorithms can be naturally adapted to adversarial situations, where a min–max local improvement operator like (4) used to define the value function. Algorithm 2 it extended as follows.

**Algorithm 3** (*Recursion for games*).

```
real proc Value(x)

if x = x_*
    return(0)
else
    if V⃗(x) = ∞
        forall u ∈ U
            Val_u := 0
            forall v ∈ V
                Val'_u := c(x,u,v) + V⃗(f(x,u,v))
                Val_u := max{Val_u, Val'_u}
            V⃗(x) := min{V⃗(x), Val_u}
    return(V⃗(x))
```

This algorithm is polynomial in the size of the game graph but, as noted before, this is not of much help for exponential transition graphs. This procedure can be adapted in various ways to explore a subset of the paths. Unfortunately, heuristic search algorithms are not as clean mathematical objects as are value functions, and a fully-systematic survey of all possible variants is beyond the scope of this article, and only a sketch is given.

The first step in guiding the exploration is to collect costs as we develop a path from $x_0$, associating with every partial path leading to $x$ a *cost-to-come* function $\overleftarrow{\mathcal{V}}(x)$ indicating the cost to reach $x$ from $x_0$.

The second step is to define an *estimation function* $\mathcal{E}$ on the state space such that $\mathcal{E}(x)$ is an approximation of the cost-to-go $\vec{\mathcal{V}}(x)$. The choice of the estimation function is *domain specific*. It should be much easier to compute than $\vec{\mathcal{V}}$ but should, nevertheless, give some good indication for the "goodness" of the state. For the scheduling problems discussed in the second part of this article, $\vec{\mathcal{V}}(x)$ would be the length of the optimal schedule from $x$, while $\mathcal{E}(x)$ can be, for example, the amount of remaining work in state $x$ divided by the number of machines.

The last step is to modify the exploration mechanism from depth-first to a more sophisticated search regime which keeps a

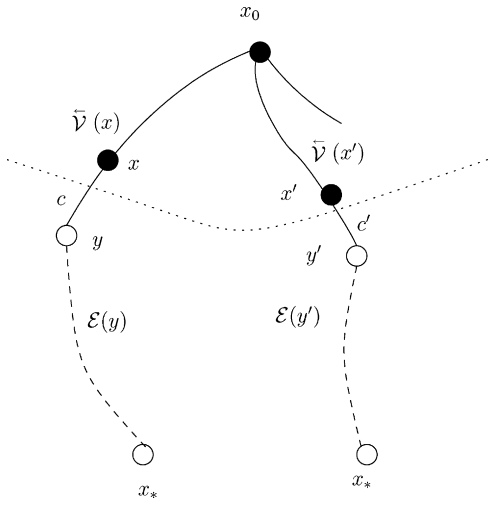


Fig. 8. A snapshot of a forward search procedure. The frontier is indicated by a dotted line. Nodes $y$ and $y'$ are compared according to $\overleftarrow{\mathrm{V}}(x) + c + \mathcal{E}(y)$ and $\overleftarrow{\mathrm{V}}(x') + c' + \mathcal{E}(y')$.

*search tree* of explored partial paths (or partial strategies) and incrementally expands this tree by exploring each step one successor $y$ of a node $x$ in its *frontier*. The criterion selecting the successor is based on the cost-to-come $\overleftarrow{\mathcal{V}}(x)$, the cost $c$ of the transitions from $x$ to $y$ and the estimation function $\mathcal{E}(y)$ (see Fig. 8).

Since the exploration is not complete, the computed value function is not exact, and the strategy is not optimal. In a game setting, it is important to distinguish between the effect of partial exploration of environment and controller actions. If we do not explore all $u$-successors, we might miss an optimal strategy. On the other hand, if we do not explore some $v$-successors we risk being too optimistic about our chosen strategy (if we use the min max criterion). Moreover, ignoring some $v$ values, the value function and the strategy might not be computed for some reachable states. In this case some rules are needed to tell the controller what to do during execution once such a state $x$ is encountered. A typical solution could be to act according to $s(x')$ where $x'$ is the nearest state for which the strategy has been computed.

It should be noted that forward exploration of reachable states is very common in verification of discrete systems but since there the performance criterion is discrete (correct/incorrect), exhaustive coverage of the adversary actions is indispensable if we want to prove correctness and hence breath-first exploration is more common. Heuristic search is sometimes used for finding bugs (bad trajectories) quickly.

## 7. On adversaries in control models

Before proceeding let me announce the resolution of a puzzle which intrigued me for some time concerning the lack of *explicit* modeling of the adversary in some of the most popular models of automatic control, which is supposed to be *the* science of combatting external disturbances. And indeed, several models of linear control mention only one player, and the controller is synthesized to be stable or optimal relative to

an adversary-free model. For an outsider like myself this seems absurd: in the discrete world if you do not have disturbances a plan without feed-back is sufficient and if you have such disturbances, plans that ignore them are rather useless.

In stability analysis one assumes that the controlled system lives around an equilibrium and the disturbance is implicitly assumed to be active only at time zero (''step response''), leading to a deviation from the equilibrium. From there, the controller is guaranteed to bring the system to the vicinity of the equilibrium. This guarantee is based on the analysis of a nominal disturbance-free model, hence avoiding the complications inherent in universal quantification on the adversary. Likewise, in LQR optimal control the best action is computed based on an adversary-free model, but since it is computed over the whole state space, it is defined over all states that an unmodeled (but small) disturbance may take you. The same reasoning underlies model-predictive control where a controller with similar properties is computed *online*. When at state $x$ a control is computed which is optimal for a bounded horizon relative to a nominal disturbance-free model, but then, in the next step where due to unmodeled disturbances the system may be at a state different from the predicted one, control is recomputed based, again, on the nominal model.

These schemes work (when they work[3]) due to a combination of factors. The first of those is probably related to continuity and linearity combined with the small magnitude of the disturbance, which guarantee that stability is preserved under disturbances and optimality is not seriously affected. This is not the case for discrete and hybrid systems (and probably even for continuous nonlinear ones) where disturbances can have a much larger impact. The second factor is that by defining the strategy all over the state space, provisions are taken for a possible deviation of the plant from its nominal model. All in all, this peculiar way of doing things simplifies computations that could be much more difficult using a full game model.

This concludes the first part of the paper in which three approaches to solve optimal control problems in the presence of an adversary were presented. The first approach was based on bounded horizon and finite dimensional optimization. The two other approaches were based on propagation of costs along paths, either backward (dynamic programming) or forward. These approaches were described using discrete $U$ and $V$ and their adaptation to continuous domains is not straightforward, unless they are discretized. In the following sections we demonstrate this approach on an interesting type of a game played with discrete values over continuous time, namely, scheduling under uncertainty in task durations.

## 8. Scheduling as a game

Scheduling problems appear in diverse situations where the use of bounded resources over time has to be regulated. A scheduler is a mechanism that decides at each time instant whether or not to allocate a resource to one of the tasks that needs it. Unfortunately, scheduling research is spread over many application domains, and in many of them problems are often solved using domain specific methods, without leading to a more general theory (except for, perhaps, operations research where scheduling is treated as a static optimization problem, similar to the approach described in Section 4). In this section we will reformulate scheduling in our terminology of dynamic two player games.

On one side of the problem we have the *resources*, a set $M = \{m_1, \ldots, m_k\}$ of ''machines'' that we assume to be fixed. On the other side we have *tasks*, units of work that require the allocation of certain machines for certain durations in order to be accomplished. In a world of unbounded resources scheduling is not a problem: each task picks resources as soon as it needs them and terminates at its earliest convenience. When this is not the case, two tasks may need the same resource at the same time and the scheduler has to resolve the conflict and decide to whom to give the resource first. The tasks may be related to each other by various inter-dependence conditions, the most typical among them is *precedence*: a task can start only after some other tasks (its predecessors) have terminated. In this paper we assume the set of tasks to be fixed and known in advance.

To model such situations as dynamic games we need first to fix the state-space. For our purposes we take the state of the system at any given instant to include the states of the tasks (waiting, active, finished), the time already elapsed (for active tasks) and the corresponding states of the machines (idle, or busy when it is used by an active task). The actions of the scheduler are of two types, the first being actions of the form *start(p)* which means allocating a machine $m$ to task $p$ so that it can execute. The effect of such an action on a state where $p$ is enabled (all its predecessors have terminated) and $m$ is idle, is to make $p$ active and $m$ occupied. Let us denote this set of actions by $S$. The other ''action'' of the scheduler is to do nothing, denoted by $\perp$. In this case the active tasks continue to execute, the waiting tasks keep on waiting and time elapses. The actions of the environment consist of similar waiting and a set of actions of the form *end(p)* whose effect, when the task spent enough time in an active state, is to move the task to a terminal state and release the machine. We assume that the environment is deterministic, that is, every *end(p)* transition occurs exactly $d$ time after the *start(p)* where $d$ is the pre-specified duration of the task (later, we will relax this assumption). In this case the strategy can be viewed as a single schedule, a function $s : \mathbb{R}_+ \to S \cup \{\perp\}$. For all but a finite number of time instances we have $s(t) = \perp$ and the schedule is determined by a finite number of start times for each task.

## 9. Deterministic job shop scheduling

A job shop problem consists of a finite set $J = \{J^1, \ldots, J^n\}$ of jobs to be processed on a finite set $M$ of machines. Each job $J^i$ consists of 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 duration $d$. Each machine can

[3] Airplanes fly, after all.

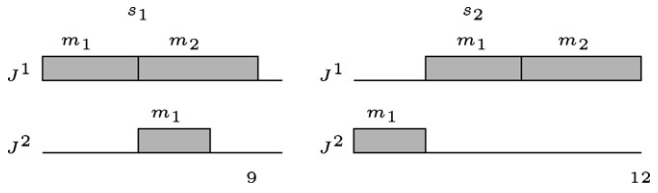


Fig. 9. Two schedule $s_1$ and $s_2$ for the example.

process at most one task at a time and, due to precedence constraints, at most one task of each job can be active at any time. Tasks cannot be preempted once started. We want to determine the starting times for each task so that the total execution time of all jobs (the time the last task terminates) is minimal.

As an example consider the problem

$$J^1 : (m_1, 4), (m_2, 5), \qquad J^2 : (m_1, 3)$$

which exhibits a conflict on $m_1$. This conflict can be resolved in two ways, either by giving priority to $J^1$ or to $J^2$ (schedules $s_1$ and $s_2$ of Fig. 9). The length induced by $s_1$ is 9 and it is the optimal schedule for this example. Part of the hardness of the problem stems from the fact that sometimes an optimal schedule is achieved by *not* executing a task as soon as it is ready in order to keep the machine free for another, still not enabled, task that will need it in the future.

The traditional way to solve this problem is to assign variables, $z_1$, $z_2$ and $z_3$ for the start times of the three tasks, a variable $z_4$ for the total length of the schedule and solve a constrained optimization problem. The precedence constraints for $J^1$ are expressed by $z_2 \geq z_1 + 4$ (it cannot use $m_2$ before finishing using $m_1$). The fact that only one task can use $m_1$ at a given time is expressed by the condition

$$[z_1, z_1 + 4] \cap [z_2, z_2 + 3] = \varnothing$$

stating that the utilization periods of $m_1$ by both jobs should not coincide. The whole problem is thus formulated as:

$$\begin{aligned}
&\min (z_4) \text{ subject to}\\
&z_2 - z_1 \geq 4,\\
&z_4 - z_2 \geq 5,\\
&z_4 - z_3 \geq 3\\
&(z_2 - z_1 \geq 4 \vee z_1 - z_2 \geq 3)
\end{aligned}$$

The format of this problem is both simpler and more complex than general linear programming. On one hand the constraints are always of the form $z_i - z_j \geq d$ rather than arbitrary linear inequalities. On the other, the last disjunctive constraint, which expresses a discrete choice, makes the set of feasible solutions *non-convex*. As the problem gets larger, the set of feasible solutions gets more and more fragmented into a disjoint union of convex polyhedra whose number is exponential in the number of conflicts. Like many other combinatorial optimization problems, job shop scheduling is NP-hard and this suggests that any algorithm might, in some cases, end up enumerating all possible solutions.

It is worth mentioning that people accustomed to continuous optimization tend to transform the problem into mixed integer-linear program by introducing auxiliary integer variables with which it is possible to encode disjunctions as arithmetical constraints. The problem is then transformed via relaxation (assuming temporarily that these variables are real-valued) into a convex linear program which can be solved efficiently. Then it remains to transform the obtained "solution" to a feasible solution with integer values for the relaxed variables. While this approach has been reported to work well for some classes of problems, I have doubts concerning its usefulness for scheduling, a problem dominated by discrete choices that *have no numerical interpretation*.

## 10. Scheduling with timed automata

In this section I sketch in more detail the modeling of scheduling situations as a dynamical system on which optimal paths and optimal strategies can be computed using the forward search algorithm discussed in Section 6. We use the timed automaton model which has established itself as the formalism of choice for describing discrete time-dependent behaviors. Timed automata are automata operating in the dense time domain. Their state-space is a product of a finite set of discrete states (locations) and the clock-space $\mathbb{R}^m_+$, the set of possible valuations of clock variables. The behavior of the automaton consists of an alternation of time-passage periods where the automaton stays in the same location and the clock values grow uniformly, and of instantaneous transitions that can be taken when clock values satisfy certain conditions and which may reset some clocks to zero. The interaction between clock values and discrete transitions is specified by conditions on the clock-space which determine what future evolution, either passage of time or one or more transitions, is possible at a state.

When timed automata model scheduling problems, the discrete states record the qualitative state of the scheduling problem (who is executing, who has terminated) and the clocks provide the quantitative component of the state, namely the times that each active task has already spent executing. We assume here that there is a single machine of each type and hence the states of the machines are implied by the states of the tasks. We will spare from the reader the exact formal definition of timed automata and illustrate our modeling approach via an example. A more formal treatment can be found in (Abdeddaïm, Asarin, & Maler, 2006).

We start by modeling each job as a simple automaton with one clock. The automata for our example, depicted in Fig. 10, have a straightforward structure. Automaton $\mathcal{A}_1$ starts with state $\bar{m}_1$ where it waits for machine $m_1$. It stays at this state until a transition to active state $m_1$ is taken. This "start" transition is issued by the scheduler and it resets clock $c_1$ to zero. The automaton stays at that state until the clock reaches 4 and then moves to state $\bar{m}_2$, waiting for the next task and so on until it reaches a final state. The "end" transitions outgoing from active states are made by the environment and are considered as actions uncontrolled by the scheduler. Clocks are considered "inactive" at waiting states as they are reset to zero before they are tested.

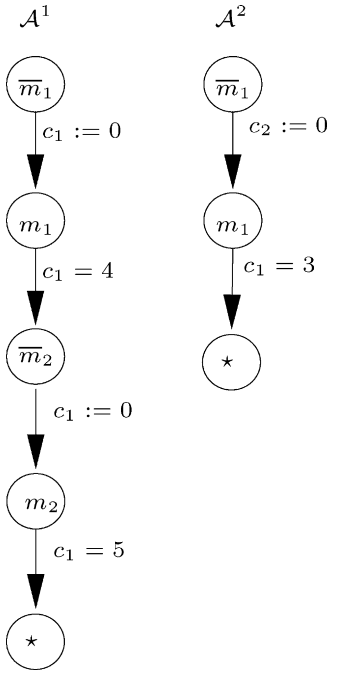


Fig. 10. Automata for the two jobs.

Each automaton describes the possible behaviors of one job in isolation. Their joint behavior under resource constraints is captured by their product shown in Fig. 11. Products of automata underly almost everything one wants to say about the behavior of several interacting components. A state space of the product (or the *global* state space) is essentially a subset of the Cartesian product of the local state spaces, and the transitions outgoing from a global state are defined based on transitions outgoing from each component.[4] Resource constraints are expressed by excluding states such as $(m_1, m_1)$ where more than one job uses a machine. This results in a "hole" in the global automaton and the scheduler has to decide how to bypass this hole, either by giving the machine first to $J^1$ or to $J^2$. The two schedules of Fig. 9 correspond to the following two behaviors (runs) of the automaton (we use notation $\perp$ to indicate inactive clocks, and $\stackrel{0}{\rightarrow}$ for discrete actions such as starting or ending a task):

$$
\begin{aligned}
s_1 : {} & (\bar{m}_1, \bar{m}_1, \perp, \perp) \stackrel{0}{\rightarrow} (m_1, \bar{m}_1, 0, \perp) \stackrel{4}{\rightarrow} (m_1, \bar{m}_1, 4, \perp) \stackrel{0}{\rightarrow} \\
& (\bar{m}_2, \bar{m}_1, \perp, \perp) \stackrel{0}{\rightarrow} (\bar{m}_2, m_1, 0, \perp) \stackrel{0}{\rightarrow} (m_2, m_1, 0, 0) \stackrel{3}{\rightarrow} \\
& (m_2, m_1, 3, 3) \stackrel{0}{\rightarrow} (m_2, \star, 3, \perp) \stackrel{2}{\rightarrow} (m_2, \star, 5, \perp) \stackrel{0}{\rightarrow} \\
& (\star, \star, \perp, \perp) \\
s_2 : {} & (\bar{m}_1, \bar{m}_1, \perp, \perp) \stackrel{0}{\rightarrow} (\bar{m}_1, m_1, \perp, 0) \stackrel{3}{\rightarrow} \\
& (\bar{m}_1, m_1, \perp, 3) \stackrel{0}{\rightarrow} (\bar{m}_1, \star, \perp, \perp) \stackrel{0}{\rightarrow} (m_1, \star, 0, \perp) \stackrel{4}{\rightarrow} \\
& (m_1, \star, 4, \perp) \stackrel{0}{\rightarrow} (\bar{m}_2, \star, \perp, \perp) \stackrel{0}{\rightarrow} (m_2, \star, 0, \perp) \stackrel{5}{\rightarrow} \\
& (m_2, \star, 5, \perp) \stackrel{0}{\rightarrow} (\star, \star, \perp, \perp)
\end{aligned}
$$

[4] In the context of linear system and Markov chains, the product of automata is sometimes disguised as a Kronecker or Tensor product of two matrices, but the concepts developed in computer science are much richer and can express a variety of interaction modes between components, ranging from independence to strong synchronization.

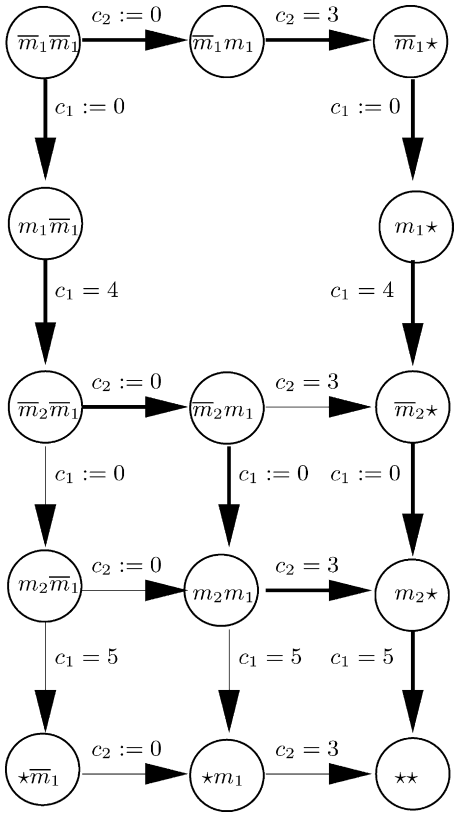


Fig. 11. The global timed automaton for the two jobs. The paths that correspond to the two schedules are indicated by thicker arrows.

It is not hard to see the correspondence between the set of possible behaviors of the automaton that reach the final state and the set of all feasible schedules. Hence the problem of optimal scheduling reduces to finding the shortest run in a timed automaton, where the length of the run is the total elapsed time. The number of such runs is uncountable (each automaton may stay any amount of time in a waiting state) however we have shown that the optimum is found among a *finite* number of runs and each node in the search tree has a finite number of successors worth exploring. The number of such paths is still exponential and an exhaustive search is infeasible. Our implementation of a best-first search algorithm on this model could find optimal schedules for problems with 6 jobs, 6 machines and 36 tasks. Beyond that we had to apply a heuristic that could find solutions with 5% from the known optimum for problems with up to 15 jobs, 15 machines and 225 tasks.[5]

The reader probably noticed that the dynamic model used here does not fit exactly into the discrete time synchronous framework previously described. By using a "sampled" approach and restricting events to occur and to be observed only at multiples of some constant $\delta$, we can approximate any timed automaton by a discrete time system. However, when events occur sparsely over time, the continuous time asynchronous approach is computationally more efficient as

[5] The coincidence between the number of jobs and machines is specific to the used benchmarks and has no deeper meaning.

it allows to "accelerate" the evolution of the system by letting time advance until the next event.

## 11. Scheduling under uncertainty

Although the approach just described is elegant, one may argue that the world of scheduling could live without yet another technique for solving the job shop problem. The advantage of using state-based dynamic models is manifested when we move to the more complex problems of scheduling under uncertainty. Academic scheduling research has often been criticized from a practical point of view for making unrealistic assumptions and it was noted that real schedules are rarely executed as planned. During execution it may happen that tasks terminate sooner or later then expected, new tasks may appear, machines may break down, etc. In such situations what we need is a scheduling policy, a strategy which adapts to the evolution of the plant and modifies its decisions accordingly. In this section we augment the job shop problem with one type of uncertainty, namely bounded uncertainty in task durations. This means that a task description gets the form $(m, [l, h])$ indicating that the actual duration of the task is some $d \in [l, h]$. Each actual *instance* of the job shop problem consists of picking such a $d$ for each interval and we need to evaluate a strategy according to its performance on all such instances. Consider the problem

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

where the only resource under conflict is $m_3$ and the order of its utilization is the only decision of the scheduler. 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 $d = (d_2, d_3) \in [2, 8] \times [2, 4]$. Fig. 12(a) depicts the optimal schedules for the instances (8, 4), (8, 2) and (4, 4) that could have been found by a non-causal *clairvoyant* scheduler who knows the whole instance in advance. But instances reveal themselves *progressively* during execution—the value of $d_1$, for example, is known *only after the termination* of the second task of $J^1$.

It turns out that for this particular type of uncertainty, optimization with respect to the worst-case criterion is somewhat trivial. There is always a maximal (critical) instance, (8, 4) in this example, having two important properties: (1) the optimal schedule for this instance is valid also for all other smaller instances (just ignore earlier termination of certain tasks and keep the machine busy until $h$ time elapses); (2) no strategy can perform better on this instance. Fig. 12(b) shows the behavior of a static worst-case strategy based on instance (8, 4) and one can see that is rather wasteful for other instances. We want a smarter adaptive scheduler which takes the actual duration of $m_2$ into consideration.

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 an assignment of absolute start times to tasks, we extract from it only the *qualitative information*, the order in which conflicting tasks utilize each resource. In our example the optimal schedule for instance (8, 4) is associated with giving priority to $J^1$ on $m_3$. Then, during execution, we start every task as soon as its predecessors have terminated, provided that the ordering is not violated. As Fig. 12(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.

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 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.

The hole filling strategy is optimal for all instances whose optimal schedule has the same ordering as that for the nominal instance. It is not good, however for instances such as (4, 4) which cannot benefit from the early termination of $m_2$ because shifting $m_3$ of $J^2$ forward will violate the priority on $m_3$. For such cases a more refined form of adaptiveness is required. Looking at the optimal schedules for (8, 4) and (4, 4) in Fig. 12(a), we observe 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 *quantitative* information encoded by clock values.

Consider the following approach: initially we find an optimal schedule for some nominal instance. During execution, whenever a task terminates we reschedule the "residual" problem, assuming nominal duration for 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

$$J'_1 : (\mathbf{m_1}, \mathbf{6}), (m_3, 4), (m_4, 5), \qquad J'_2 : (m_3, 7)$$

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 give $m_3$ to $J^2$. Likewise, if $m_2$ terminates at 8 we have

$$J'_1 : (\mathbf{m_1}, \mathbf{2}), (m_3, 4), (m_4, 5), \qquad J'_2 : (m_3, 7)$$

and the optimal schedule consists of waiting for the termination of $m_1$ in order to give $m_3$ to $J^1$. The property of schedules thus obtained is that at any state reachable during execution they are optimal with respect to the nominal assumption concerning the *future*. We call such strategies *d-future optimal*.

This is the principle underlying model-predictive control where at each step, actions at the current "real" state are re-optimized while assuming some nominal prediction for a bounded horizon future. A major drawback of this approach is that it involves a lot of *online* computation, solving a new scheduling problem each time a task terminates. This fact restricts its applicability to "slow" processes. In the next section we present an alternative approach where an equivalent

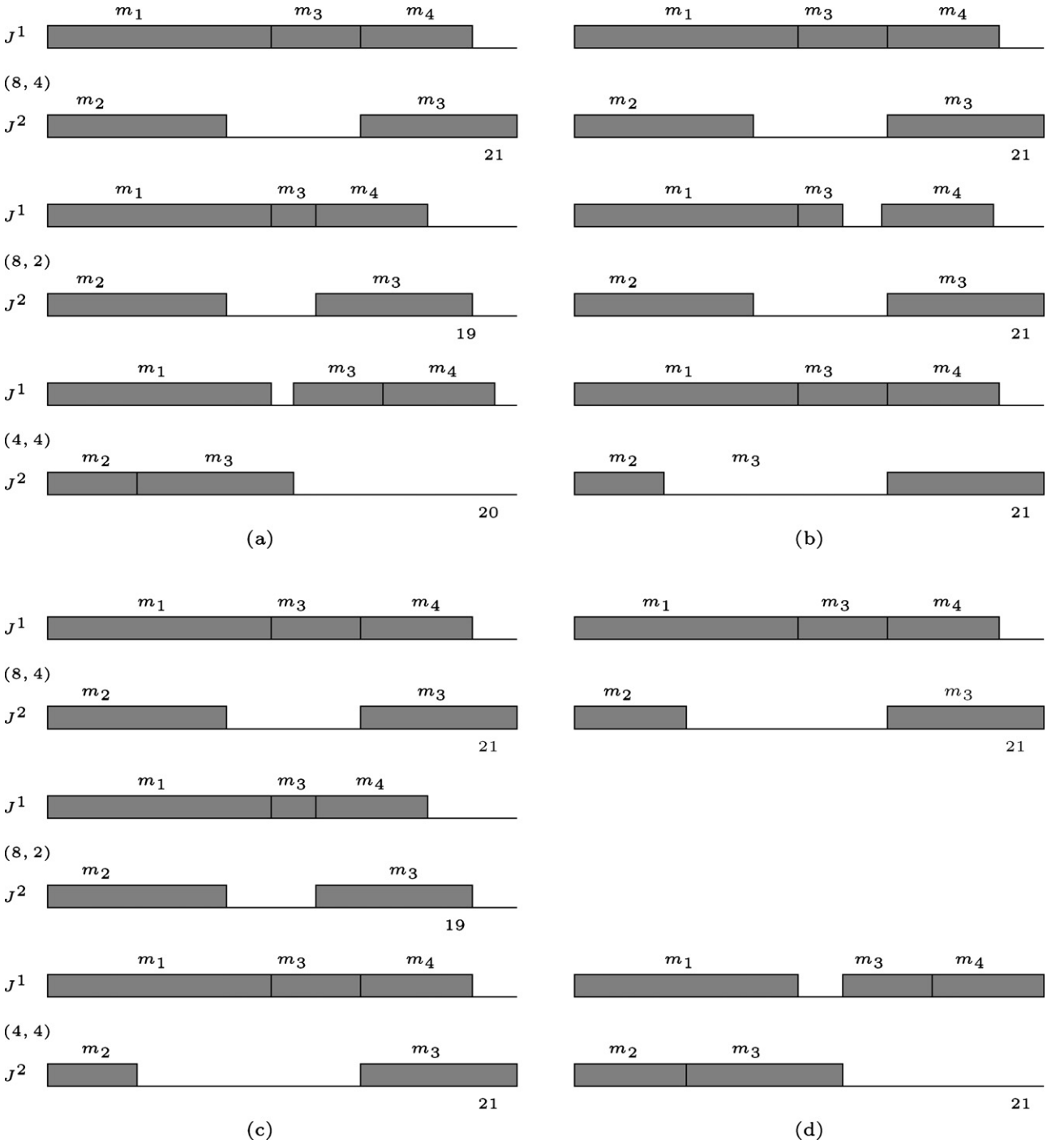


Fig. 12. (a) Optimal schedules for three instances; (b) a static schedule based on the worst instance (8, 4); (c) the behavior of a hole filling strategy based on instance (8, 4); (d) the equal performance of the two strategies on instance (5, 4).

strategy is synthesized *offline* using a symbolic variant of dynamic programming adapted for timed automata.

## 12. Dynamic programming on timed automata

The state-space of a timed automaton consists of pairs of the form $(q, c)$ where $q = (q_1, \ldots, q_n)$ is a discrete state, indicating the local states of all jobs, and $c = (c_1, \ldots, c_n)$ is a vector of clock valuations ranging over a bounded subset of the non-negative reals. On these we define a value function $\vec{\mathcal{V}}$ such that $\vec{\mathcal{V}}(q, c)$ denotes the minimal time to reach the final state from $(q, c)$, assuming nominal values for tasks that have not terminated. Before giving the formal definition let us give an intuitive explanation. Being at $(q, c)$, all the local choices of the scheduler can be brought into the following form: *let some time pass and then execute one transition that is enabled at that time according to the clock valuest*. This definition covers also the possibility of an immediate action ($t = 0$), as well as the possibility of waiting until an uncontrolled transition is taken by the environment. The value induced by this choice is the sum of

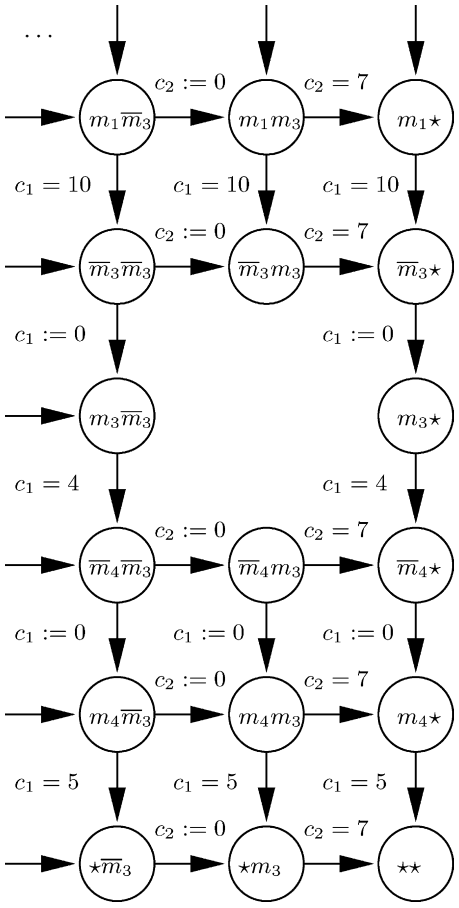


Fig. 13. Part of the global automaton.

the waiting time $t$ and the value of the state reached after the transition. This is captured by the following recursive definition:

$$\vec{\mathcal{V}}(\star, c) = 0,$$
$$\vec{\mathcal{V}}(q, c) = \min\{t + \vec{\mathcal{V}}(q', c') : (q, c) \to^{t} (q, c + t\mathbf{1}) \to^{0} (q', c')\}$$

To illustrate the computation of $\vec{\mathcal{V}}$ we consider a simplified version of the example from the previous section with only one uncertain duration:

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

Fig. 13 shows the final part of the global automaton corresponding to the problem, which includes state $(m_1, \bar{m}_3)$ where a decision of the scheduler has to be taken. The computation starts with $\vec{\mathcal{V}}(\star, \star, \perp, \perp) = 0$. The value of $(m_4, \star, c_1, \perp)$ is the time it takes to satisfy the condition $c_1 = 5$, which is $5 \dot{-} c_1$. Likewise $\vec{\mathcal{V}}(\star, m_3, \perp, c_2) = 7 \dot{-} c_2$. In state $(m_4, m_3)$ the two jobs are active and the transition to be taken depends on which of them will "win the race" and terminate before:

$$\vec{\mathcal{V}}(m_4, m_3, c_1, c_2)$$
$$= \min \begin{Bmatrix} 7 \dot{-} c_2 + \vec{\mathcal{V}}(m_4, \star, c_1 + (7 \dot{-} c_2), \perp), \\ 5 \dot{-} c_1 + \vec{\mathcal{V}}(\star, m_3, \perp, c_2 + (5 \dot{-} c_1)) \end{Bmatrix}$$
$$= \min\{5 \dot{-} c_1, 7 \dot{-} c_2\} = \begin{cases} 5 \dot{-} c_1 & \text{if } c_2 \dot{-} c_1 \geq 2 \\ 7 \dot{-} c_2 & \text{if } c_2 \dot{-} c_1 \leq 2 \end{cases}$$

Note that both transitions are uncontrolled *end* transitions and no decision of the scheduler is required in this state. The computation proceeds backwards, computing $\vec{\mathcal{V}}$ for all states. In particular, for state $(m_1, \bar{m}_3)$ where we need to choose between giving $m_3$ immediately to $J^2$ or waiting for the termination of $m_1$ to give $m_3$ to $J^1$, we obtain:

$$\vec{\mathcal{V}}(m_1, \bar{m}_3, c_1, \perp) = \min\{16, 21 \dot{-} c_1\}$$
$$= \begin{cases} 16 & \text{if } c_1 \leq 5 \\ 21 \dot{-} c_1 & \text{if } c_1 \geq 5 \end{cases}$$

Hence, if $m_1$ terminates after less than 5 time units it is better to give $m_3$ to $J^2$, otherwise it is worth waiting and giving it to $J^1$. Fig. 12(d) shows that, indeed, the two strategies coincide in performance when $c_1 = 5$.

The reader should not be misled by the success of our strategy to match the performance of a clairvoyant scheduler for this small example. In slightly more complex problems with several uncertainties it is impossible to compete with knowing the future and being $d$-future optimal is good enough.

The actual computation of the value function is implemented using standard reachability techniques for timed automata which are outside the scope of the present paper. We have tested our implementation on a problem with 4 jobs, 6 machines and 24 tasks, 8 of which having uncertain durations. we fixed two instances, one "optimistic" where each task duration is set to $l$, and one "pessimistic" with $h$ durations. We applied our algorithm to find two $d$-future optimal strategies and two hole filling strategies based on these instances. We have generated 100 random instances with durations drawn uniformly from each $[l, h]$ interval, and compared the results of the above-mentioned strategies with an optimal clairvoyant scheduler that knows each $d$ in advance, and with the static worst-case scheduler. It turns out that the static schedule is, on the average, longer than the optimum by more than 12%. The hole filling strategy deviates from the optimum by around 5%. Our strategy produces schedules that are longer than the optimum by less than 2%.

The good news is that our strategy is much better than static scheduling, and can be considered as a useful tool for systems with "soft" real-time performance criteria. The bad news is that it is much more costly than the hole filling strategy. The latter solves an adversary-free problem and can use intelligent forward search while the computation of our strategy has to explore the whole state-space. The adaptation of forward game tree search to this problem is not straightforward, due to the density of the set of adversary actions, and it is subject to ongoing research along with the adaptation of this approach to other types of uncertainty such as imprecise arrival times or discrete uncertainty associated with conditional dependencies between tasks.

## 13. Discussion

People who are experts in their domain are often skeptical toward proposals for unified theories. Indeed, compared to successes of domain specific research, various holistic trends

such as ''general systems theory'' proved in the past to be rather sterile. Saying that ''everything is systems'' and that many things that look so different are, at a certain level of abstraction, similar, does not necessarily solve problems. I hope that the framework presented in this paper will have a better fate. It is less ambitious than some of its predecessors in the sense of not trying to predict the unpredictable and pretend to give optimal recipes for complex socio-economic or biological phenomena for which we do not even know the appropriate modeling vocabulary. Rather it is restricted to situations where useful dynamic models and performance criteria do exist, models which are already used, implicitly or explicitly, for simulation, verification or optimization. This framework is geared toward a *concrete* goal: developing a tool for defining and solving optimal control problems for systems with diverse types of dynamics.

Some principles underlying such a framework (some of which already exist in respective domains) are mentioned below. First, I believe that systems should be defined with a clear semantics from which it is easy to see who are the players, what are the variables they can observe and influence, what constitutes a behavior of the system, what is assumed about the environment and what are the natural performance criteria. At this level, the description should be *separated* from the specific computational techniques that are used to reason about the model. This is in contrast with some domain-specific approaches where problems are often phrased in terms biased toward particular and, sometimes, accidental solution techniques which are common in the domain.

After an ideal optimal controller has been mathematically defined, computational issues should be addressed. Here the difference between classes of system dynamics is manifested by the type of constrained optimization problem to be solved, discrete (logical), continuous (numerical) or hybrid. In most cases the global optimality of the solution is a ceremonial matter. No one really intends to be optimal and models are imprecise anyway. In some cases, proving some relation between approximate solutions and the optimum is a good measure for the quality of a technique, but this is neither a necessary nor a sufficient condition for its usefulness.

Since some space is left, let me add some controversial remarks. It seems to me that in many domains relevant to this paper, there is a tension between the mathematical (theoretical) and engineering (hacking) approaches. The (real) practitioner cannot choose the problems he has to solve and also does not have time to develop nice theories. In many cases he will adapt solutions provided by mathematicians of previous generations to get the job done. The theoretician is supposed to be more open-minded and explore new classes of models for new phenomena but the structure of academe does not always encourage him to do so. Members of scientific communities often impose upon themselves some intrinsic evaluation criteria that deviate over time from the raison d'être of the domain. There is nothing wrong with (good) mathematics for its own sake, but one should not confuse it with solving real engineering problems or even with laying the foundations for future solutions. What is really needed is a middle road between mathematics and engineering, which allows us to see the generic mathematical objects behind the engineering instances, together with a *strong sense of criticism* toward the traditions of the respective academic fields, which are often by-products of the sociology of scientific communities, rather than the result of a genuine attempt to be relevant.

## Acknowledgments

Any attempt to give a fair survey of related work, spanning over more than 50 years and numerous scientific communities, will either be pretentious or require an effort that goes beyond the scope of this paper. Consequently, I provide no references to the first part of the paper. The results concerning scheduling were obtained in collaboration with Y. Abdeddaïm and E. Asarin. Readers interested in more details might want to look at the thesis (Abdeddaïm, 2002) or the paper (Abdeddaïm et al., 2006).

This work was inspired partly by an interaction with control theorists on various occasions related to hybrid (discrete/continuous) systems, including my partners in the CC project, whom I would like to thank (the opinions are, of course, my own). The work on scheduling enjoyed similar relations with partners in the AMETIST project. This manuscript benefited from feed-back provided by Anil Nerode, George Pappas, Anders Ravn, Tariq Samad, Jan van Schuppen, Michel Sintzoff and Pravin Varaiya, and from numerous past discussions with Eugene Asarin and Bruce Krogh.

## References

Abdeddaïm, Y. (2002). *Scheduling with timed automata*. PhD Thesis, Institut National Polytechnique de Grenoble, Laboratoire Verimag.

Abdeddaïm, Y., Asarin, E., & Maler, O. (2006). Scheduling with timed automata. *Theoretical Computer Science, 354*, 272–300.

**Oded Maler** is a senior researcher (''Directeur de Recherche'') at the CNRS (French national research council) and is located at the VERIMAG Laboratory in Grenoble where he is responsible for the *timed and hybrid systems* group. He holds a B.A. degree from the Technion, Haifa in computer science, an M.Sc. from Tel-Aviv University in management science and a Ph.D. from Weizmann Institute, Rehovot, in computer science. His main research agenda is concerned with the exportation toward other domains – most notably control theory, scheduling, timing analysis and systems biology – of automata-theoretic analysis and synthesis techniques used in verification. Dr. Maler has been one of the initiators of hybrid systems research and made significant contribution to the algorithmic analysis of continuous and hybrid systems.