ON OPTIMAL AND SUB-OPTIMAL CONTROL IN THE PRESENCE OF ADVERSARIES [1]

Oded Maler *

* CNRS-Verimag, *2, av. de Vignate, 38610, Gières, France.*
`Oded.Maler@imag.fr`
`www-verimag.imag.fr/∼maler/`

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.

1. INTRODUCTION

I would like to use this opportunity to present a paper free of any "new original results", not pretending to put something new under the sun. I will describe what I consider to be the essence of many activities concerned with the design of systems based on mathematical models, without, I believe, saying anything that is not known in some of the many disciplines and communities that occupy themselves with these issues. Yet, 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.

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. We start in Section 2 with a general discussion on how to *define* the performance of a system which is subject to external disturbances. In section 3 we introduce the generic model that we use, a dynamic "multi-stage" game between a controller and its environment. 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. The classical solution technique known as dynamic programming is described in Section 5 followed, in section 6, by the alternative method of heuristic forward search.

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 7 attempts to convince the reader that scheduling falls into the class of problems described in the first part. The job shop problem is described in Section 8 along with its traditional solution scheme based on non-convex "combinatorial" optimization. In Section 9 we show how this problem can be solved using shortest path algorithms on timed automata. Section 10 extends the problem by considering bounded uncertainty in task durations, while Section 11 sketches a dynamic programming algorithm that can find adaptive strategies that are better than worst-case strategies.

The topics discussed in this paper are treated by different disciplines and under diverse titles such as System Verification, Controller Synthesis, Sequential Decision Making, Game Theory, Markov Decision Processes, AI Planning, Optimal and Model Predictive Control, Shortest Path Algorithms, Dynamic Programming, Optimization, Differential Games and more, each with its own terminology. For example, what is called in one

[1] 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).

context a strategy, can be called elsewhere a policy, a feed-back law, a controller or a dispatching rule. 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 specific terminology. I also tried not to bias the description toward my own automata-theoretic background.

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 predefined criterion, while the environment, unless one is paranoid, is indifferent to the results. To illustrate the problematics of optimizing something that depends not only on one's own actions we start with 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. There are basically three generic approaches for evaluating our decisions:

- *Worst-case*: each action of the controller is evaluated according to the worst outcome that may result from taking the action:
$$u = argmin\ \max\{c(u, v_1), c(u, v_2)\}.$$
- *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:
$$u = argmin\ p(v_1) \cdot c(u, v_1) + p(v_2) \cdot c(u, v_2).$$
- *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 denying the existence of uncontrolled disturbances and the problem is reduced to ordinary optimization:
$$u = argmin\ c(u, v_1).$$

Before moving to dynamic games let us note what happens when c is a continuous function over continuous U and V. The average case analysis stays within the standard framework of continuous optimization, that is, optimization of a real-valued function, while the worst-case min-max analysis does not. This may partially explain why in domains such as continuous control stochastic disturbances are much more popular.

3. DYNAMIC GAMES

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. A game is characterized by a state-space X and a dynamic rule of the form
$$x' = f(x, u, v)$$
stating that at each time instant the "next" value of 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 such a dynamics is often written as a recurrence equation of the form
$$x_i = f(x_{i-1}, u_i, v_i)$$
but we keep in mind the existence of other models such as differential games on dense time defined via
$$\dot{x} = f(x, u, v)$$
or games with a more "asynchronous" flavor where actions may occur on a non-periodic time set (event-triggered rather then time-triggered) and where actions of both players need not occur simultaneously. Such asynchronous games will be used later to model scheduling problems.

We assume all games to start from an initial state x_0 and use the notation $\bar{x}$ for a state-sequence $x[0], x[1], \ldots, x[k]$. Likewise, we will use $\bar{u}$ and $\bar{v}$ for sequences of players actions. The predicate (constraint) $B(\bar{x}, \bar{u}, \bar{v})$ denotes the fact that $\bar{x}$ is the behavior of the system when the two players apply the action sequences $\bar{u}$ and $\bar{v}$, respectively:
$$\begin{aligned} B(\bar{x}, \bar{u}, \bar{v})\ \textit{iff}\ & x[0] = x_0 \\ & x[t] = f(x[t-1], u[t], v[t])\ \forall t \end{aligned}$$

It is sometimes useful to view the game as a *labeled directed graph* whose nodes are the elements of X and its edges are all the pairs (x, x') such that $f(x, u, v) = x'$ for some u and v. The reachable part of this transition graph is the subgraph obtained by restriction to elements of X for which a path from x_0 exists. An alternative useful notation for $B(\bar{x}, \bar{u}, \bar{v})$ is:

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

There are many ways to assign performance measures to such behaviors in order to compare them and find the optimal one. For example, any cost function $c : X \to \mathbb{R}$ on states can be lifted to a cost function on sequences by letting

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

or

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

Another useful measure is the minimal time to reach a goal set $F \subseteq X$:

$$c(\bar{x}) = \min\{t : x[t] \in F)\}.$$

In the more general case the cost function may also take into account the costs associated with the controller actions $\bar{u}$. The nature of cost functions depends on the application domain. In discrete verification $c(x)$ is typically a $\{0, 1\}$-valued function indicating bad states, and checking invariance properties (whether the system always avoids those bad states) reduces to checking whether $c(\bar{x}) = 0$ for the max-extension of c to sequences. In continuous domains some quadratic functions on $\bar{x}$ or other "norms" are often used to indicate the distance of the sequence (trajectory) from a reference. 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.

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 becomes 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

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

Here we used a cost function based only on $\bar{x}$ while the fact that $\bar{x}$ is the result of following the dynamics f under control $\bar{u}$ is part of the *constraints*. For linear dynamics, specified by $x' = Ax + Bu$, and a linear cost function, the problem reduces to standard linear programming. In discrete verification where the dynamics and cost are defined logically, the problem reduces to Boolean satisfiability (this is the essence of bounded model checking).

If we dispose of a constrained optimization procedure for the domain in question, we can compute the desired $\bar{u}$. Note that in the absence of external disturbances $\bar{u}$ completely determines $\bar{x}$ and no feed-back from x is needed. The control "strategy" reduces to an open-loop "plan": at each time instant t apply the element $u[t]$ of $\bar{u}$. We could have rephrased it 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 re-introduce the adversary and use, without loss of generality, the worst-case criterion. We now need to find $\bar{u}$ which is the solution of

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

Consider the case where $U = \{u_1, u_2\}$ and $V = \{v_1, v_2\}$ which is captured, for horizon 2, by the game tree of Figure 1. We assume, for simplicity, that the cost of each behavior is determined by its terminal state. In this case we can enumerate all the 4 possible control sequences and compute the cost they induce as:

$$\begin{aligned}
u_1u_1 &: \max\{c(x_5), c(x_6), c(x_9), c(x_{10})\} \\
u_1u_2 &: \max\{c(x_7), c(x_8), c(x_{11}), c(x_{12})\} \\
u_2u_1 &: \max\{c(x_{13}), c(x_{14}), c(x_{17}), c(x_{18})\} \\
u_2u_2 &: \max\{c(x_{15}), c(x_{16}), c(x_{19}), c(x_{20})\}
\end{aligned}$$

The sequence which minimizes these values is the optimal open-loop control that can be achieved. 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 $u[1] = u_1$ and we need to choose $u[2]$. If, for example, $\max\{c(x_5), c(x_6)\} < \max\{c(x_7), c(x_8)\}$ but $\max\{c(x_9), c(x_{10})\} > \max\{c(x_{11}), c(x_{12})\}$ then the optimal thing to do is to apply u_1 when $x[1] = x_1$ and u_2 when $x[1] = x_2$.

A *control strategy* is thus a function $s : X \to U$ telling the controller what to do at any reachable state of the game. The following predicate indicates the fact that $\bar{x}$ is the behavior of the

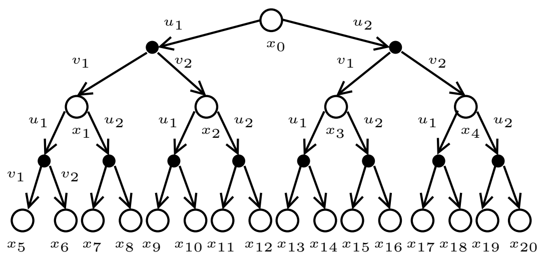
u_1
u_2
x_0
v_1
v_2
v_1
v_2
u_1
x_1
u_2
u_1
x_2
u_2
u_1
x_3
u_2
u_1
x_4
u_2
v_1
v_2
x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15} x_{16} x_{17} x_{18} x_{19} x_{20}

Fig. 1. A game tree of depth 2.

system in the presence of disturbance $\bar{v}$ when the controller employs strategy s:

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

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

$$\min_s \max_{\bar{v}} c(\bar{x}) \text{ subject to } B(\bar{x}, s, \bar{v}).$$

Finding an optimal strategy is usually much more difficult than finding an optimal sequence. Discrete finite-state systems admit $|U|^{|X|}$ potential strategies and each of them induces $|V|^k$ behaviors of length k. In continuous domains (on continuous time), where the enumeration of all strategies is not an option, such a strategy is the solution of a partial differential equation known as the Hamilton-Jacobi-Bellman-Isaacs equation. Note that a strategy need not be defined all over X, only for *elements reachable from* x_0 when the controller employs that strategy.

5. DYNAMIC PROGRAMMING

Dynamic programming, or backward value iteration, is a technique, advocated by Bellman, for computing optimal strategies in an incremental way. For discrete systems the algorithm is polynomial in the size of the transition graph, which is better than the exponential enumeration of strategies. However, as we will see later, this is not of much comfort in many situations where the transition graph itself is *exponential* in the number of system variables.

We will illustrate dynamic programming on the following shortest path problem. A subset F of X is designated as a target set, and a cost $c(x, u, v)$ is associated with each transition. The cost of a path

$$x[0] \xrightarrow{u[1],v[1]} x[1] \cdots \xrightarrow{u[k],v[k]} x[k]$$

from the initial state to a target state is

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

and our goal is to find the strategy that minimizes the worst-case.

Dynamic programming uses an auxiliary function (value function, cost-to-go) $\overrightarrow{\mathcal{V}}: X \to \mathbb{R}$ such that $\overrightarrow{\mathcal{V}}(x)$ is the performance of the optimal strategy for the sub-game starting from x. For "leveled" acyclic transition graphs (where all paths that reach a state x from x_0 have the same number of transitions), $\overrightarrow{\mathcal{V}}$ admits the following backward recursive definition:

$$\overrightarrow{\mathcal{V}}(x) = 0 \qquad \text{when } x \in F$$

$$\overrightarrow{\mathcal{V}}(x) = \min_u \max_v (c(x, u, v) + \overrightarrow{\mathcal{V}}(f(x, u, v))).$$

In more general settings, including cycles, the value function is the fixed-point of the following iteration:

$$\overrightarrow{\mathcal{V}}_0(x) = \begin{cases} 0 & \text{when } x \in F \\ \infty & \text{when } x \notin F \end{cases}$$

$$\overrightarrow{\mathcal{V}}_{i+1}(x) = \min \left\{ \begin{array}{l} \overrightarrow{\mathcal{V}}_i(x), \\ \min\limits_u \max\limits_v (c(x, u, v) + \overrightarrow{\mathcal{V}}(f(x, u, v))) \end{array} \right\}$$

We recall that the choice of max and summation in this "local" operator is specific to this particular performance criterion. When max is replaced by weighted sum, we obtain the solution procedure for Markov decision processes, leading to a strategy with optimal expected value. When summation is replaced by max, you obtain essentially the backward synthesis algorithm for discrete event systems (automata). In this case, the value of $\overrightarrow{\mathcal{V}}_i$ stands for the characteristic function of the set of states from which the controller cannot postpone reaching a forbidden state for more than i steps.

This elegant procedure is guaranteed (if it converges) to find the optimal value $\overrightarrow{\mathcal{V}}(x_0)$ of the game, as well as the strategy that attains this optimum: just take for each x the u that achieves the local optimum. For finite-state systems with positive transition costs, finite convergence is guaranteed. The only deficiency of dynamic programming is the need to compute the function for too many states, sometimes states that are not reachable from x_0 at all, and sometimes states that are not reachable by any reasonable strategy. This prevents the straightforward application of the algorithm to systems having a huge state-space (Bellman's "curse of dimensionality").

6. FORWARD SEARCH

Shortest path problems (without an adversary) admit a dual *forward* procedure, due to Dijkstra. Here we use a backward value function $\overleftarrow{\mathcal{V}}$ such that $\overleftarrow{\mathcal{V}}(x)$ indicates the minimal cost for reaching x from x_0, and we want to compute $\overleftarrow{\mathcal{V}}(x)$ for $x \in F$. This is done iteratively using:

$$\overleftarrow{\mathcal{V}}(x_0) = 0$$

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

where x' ranges over all the immediate predecessors of x. This algorithm is polynomial as well but, as noted before, this is not of much help for exponential transition graphs. The advantage of this procedure is the ability to apply intelligent search and sometimes find the optimum without exploring all the reachable states. If we do not insist on optimality, we can find reasonable solutions while exploring a small fraction of the paths.

To demonstrate this idea in a way that can be extended later for problems with adversaries, we view each incomplete path as a *partial strategy* defined only on states encountered along the path. We will store triples of the from $(s, x, \overleftarrow{\mathcal{V}})$ where s is a partial strategy, x stands for the last node in the path and $\overleftarrow{\mathcal{V}}$ is the cost for reaching x along the path. The first version of the algorithm explores all the paths (and all the partial strategies). We use a waiting list W in which we store partial paths that need to be explored further. The algorithm is given below:

```
$W := \{(\emptyset, x_0, 0)\}$
**repeat**
  Pick a non-terminal node $(s, x, \overleftarrow{\mathcal{V}}) \in W$
  **for** every $u \in U$ **do**
    $(s', x', \overleftarrow{\mathcal{V}}') :=$
          $(s \cup \{x \mapsto u\}, f(x, u), \overleftarrow{\mathcal{V}} + c(x, u))$
    Insert $(s', x', \overleftarrow{\mathcal{V}}')$ into $W$
  **end**
  Remove $(s, x, \overleftarrow{\mathcal{V}})$ from $W$
**until** $W$ contains only terminal nodes
```

If new nodes are inserted at the end of W, the graph is explored in a breadth-first manner. The nodes explored by this procedure on the example of Figure 2 appear in the table below. The four entries for the terminal nodes are compared and the path/strategy with minimal cost is selected.

s	x	$\overleftarrow{\mathcal{V}}$
$\emptyset$	x_0	0
$\{x_0 \mapsto u_1\}$	x_1	$c(x_0, u_1)$
$\{x_0 \mapsto u_2\}$	x_2	$c(x_0, u_2)$
$\{x_0 \mapsto u_1, x_1 \mapsto u_1,\}$	x_3	$c(x_0, u_1) + c(x_1, u_1)$
$\{x_0 \mapsto u_1, x_1 \mapsto u_2,\}$	x_3	$c(x_0, u_1) + c(x_1, u_2)$
$\{x_0 \mapsto u_2, x_2 \mapsto u_1,\}$	x_3	$c(x_0, u_2) + c(x_2, u_1)$
$\{x_0 \mapsto u_2, x_2 \mapsto u_2,\}$	x_3	$c(x_0, u_2) + c(x_2, u_2)$

This algorithm can be modified in order to find an optimal strategy without exhaustive exploration. The idea is to associate with every partial strategy an *estimation function* which gives a lower-bound on the cost of any extension of the strategy. This function is defined for every $(s, x, \overleftarrow{\mathcal{V}})$ in W as $\mathcal{E}(s, x, \overleftarrow{\mathcal{V}}) = \overleftarrow{\mathcal{V}} + \underline{\overrightarrow{\mathcal{V}}}(x)$ where $\underline{\overrightarrow{\mathcal{V}}}$ is an under-approximation of the cost-to-go function $\overrightarrow{\mathcal{V}}$. This function can be derived from domain-specific knowledge and it provides an optimistic estimation of the remaining cost to reach the target by any strategy that extends s. Note that as x gets deeper, the past component in $\mathcal{E}(x)$ becomes more dominant and the estimation — more realistic.

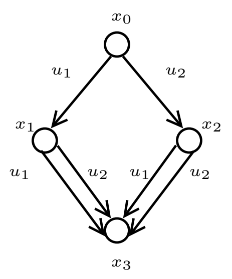

Fig. 2. A shortest path problem.

A *best-first* version of the algorithm maintains W ordered according to $\mathcal{E}$ and explores the more promising nodes first. Moreover, the algorithm can stop exploring when the value of $\mathcal{E}$ for the first element in W is larger than solutions that were already found. To increase the efficiency of the algorithm we can first perform depth-first random search to obtain some solutions at early stages of the search (branch and bound). Best-first search is guaranteed to find the optimum, and if we relax the optimality requirements, we can explore even less states, for example, by exploring only a subset of the successors of each node, or by stopping the algorithm when a solution smaller than some pre-specified value is found.

The adaptation of forward search to game graphs situations is done as follows. First let us redefine the value function as:

$$\overleftarrow{\mathcal{V}}(x) = \min_u \max_v (c(x', u, v) + \overleftarrow{\mathcal{V}}(f(x', u, v))).$$

Here, because of the adversary, every partial strategy results in a *set of states*. To avoid heavy notations we omit the cost from the reachable sets of states and the reader should keep in mind that every reachable x is, in fact, $(x, \overleftarrow{\mathcal{V}})$ where $\overleftarrow{\mathcal{V}}$ is the accumulated cost to reach x via the path in question. The set of successors of a state x via a controller action u is

$$f(x, u) = \{f(x, u, v) : v \in V\}.$$

The u-successor of (s, x) with s being a partial strategy is

$$\sigma((s, x), u)) = (s \cup \{x \mapsto u\}, f(x, u)).$$

Consider now a node of the form (s, L) where s is a partial strategy and L is the set of states reachable while following s. The successors of (s, L), that is, the partial strategies the extend s and their respective sets of reachable states, are all combinations of all possible choices of u for all $x \in L$:

$$\sigma(s,L)=\bigotimes_{x\in L}\{(\sigma(s,x),u):u\in U)\},$$

where

$$L_1\otimes L_2=\{(s_1\cup s_2,m_1\cup m_2):\\ (s_1,m_1)\in L_1,(s_2,m_2)\in L_2\}.$$

Let is illustrate this on the game tree of Figure 1. The root node, $(\emptyset,\{x_0\})$, has two successors:

$$\sigma(\emptyset,\{x_0\}) = \{\sigma((\emptyset,x_0),u_1),\sigma((\emptyset,x_0),u_2)\}$$

$$=\left\{\begin{array}{l}(\{x_0\mapsto u_1\},\{x_1,x_2\})\\(\{x_0\mapsto u_2\},\{x_3,x_4\})\end{array}\right\}$$

Let us compute the successors of the first:

$$\sigma(\{x_0\mapsto u_1\},\{(x_1,x_2\})=$$

$$\left\{\begin{array}{l}(\{x_0\mapsto u_1,x_1\mapsto u_1\},\{x_5,x_6\}),\\(\{x_0\mapsto u_1,x_1\mapsto u_2\},\{x_7,x_8\})\end{array}\right\}\otimes$$

$$\left\{\begin{array}{l}(\{x_0\mapsto u_1,x_2\mapsto u_1\},\{x_9,x_{10}\}),\\(\{x_0\mapsto u_1,x_2\mapsto u_2\},\{x_{11},x_{12}\})\end{array}\right\}$$

which gives the following four nodes

$$\left(\begin{array}{l}\{x_0\mapsto u_1,x_1\mapsto u_1,x_2\mapsto u_1\},\\\{x_5,x_6,x_9,x_{10}\}\end{array}\right),$$

$$\left(\begin{array}{l}\{x_0\mapsto u_1,x_1\mapsto u_1,x_2\mapsto u_2\},\\\{x_5,x_6,x_{11},x_{12}\}\end{array}\right),$$

$$\left(\begin{array}{l}\{x_0\mapsto u_1,x_1\mapsto u_2,x_2\mapsto u_1\},\\\{x_7,x_8,x_9,x_{10}\}\end{array}\right)$$

and

$$\left(\begin{array}{l}\{x_0\mapsto u_1,x_1\mapsto u_2,x_2\mapsto u_2\},\\\{x_7,x_8,x_{11},x_{12}\}\end{array}\right)$$

An exhaustive version of the forward search algorithm for game graphs is exponential in the size of the graph (unlike the backward procedure) but with the help of an estimation function on sets of nodes, one can hope to prune many branches of the search tree using best-first search. Note that the procedure can be adapted easily to the average-case criterion by replacing sets of states by probabilities on states.

The alert reader might have traced some inaccuracies in the description which was specialized to game trees, rather to the more general case of game graphs. In the latter case, a set of reachable states my include several copies of the same state, and all but the one with the worst cost can be removed. We also assumed that a strategy cannot reach the same state twice along the same path. A strategy that does that has an infinite value and can be discarded. For clarity of exposition we assumed that both players can apply any element of U and V at any time, while in reality some actions are possible only at certain states.

This concludes the first part of the paper in which we presented three approaches to solve optimal control problems in the presence of an adversary. 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. Discretization of U may lead to loss of optimality while discretization of V — to optimistic values of the chosen strategy. 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.

7. 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 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.

8. 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 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

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

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 Figure 3). The length induced by s_1 is 9 and it is the optimal schedule for this example. 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 task that will need it in the future.

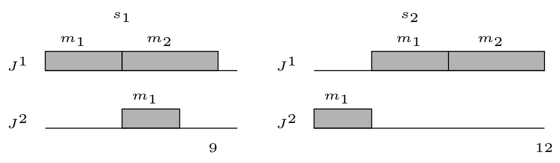

Fig. 3. Two schedule s_1 and s_2 for the example.

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

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

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

$$\begin{array}{l} \min(z_4) \quad \text{subject to} \\ \;\; z_2 - z_1 \geq 4 \\ \;\; z_4 - z_2 \geq 5 \\ \;\; z_4 - z_3 \geq 3 \\ \;\; (z_2 - z_1 \geq 4 \vee z_1 - z_2 \geq 3) \end{array}$$

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 rests 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.*

9. 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 described 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.

We start by modeling each job as a simple automaton with one clock. The automata for our example, depicted in Figure 4, have a straightforward structure. Automaton $\mathcal{A}_1$ starts with state $\overline{m}_1$ 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 $\overline{m}_2$, 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.

These automata describe the possible behaviors of each job in isolation. Their joint behavior under resource constraints is captured by their product shown in Figure 5. This is essentially a Cartesian product of the job automata, where

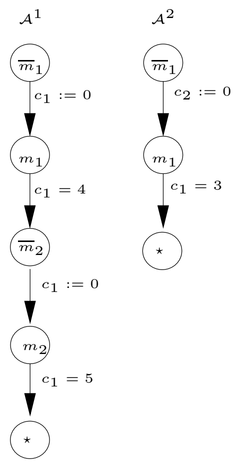

Fig. 4. Automata for the two jobs.

resource constraints are expressed by removing states such as (m_1, m_1) where more than one job uses a machine. This results in a "hole" in the 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 Figure 3 correspond to the following two behaviors (runs) of the automaton (we use notation $\perp$ to indicate inactive clocks, and $\xrightarrow{0}$ for discrete actions such as starting or ending a task):

s_1 :
$(\overline{m}_1, \overline{m}_1, \perp, \perp) \xrightarrow{0} (m_1, \overline{m}_1, 0, \perp) \xrightarrow{4} (m_1, \overline{m}_1, 4, \perp) \xrightarrow{0}$
$(\overline{m}_2, \overline{m}_1, \perp, \perp) \xrightarrow{0} (\overline{m}_2, m_1, 0, \perp) \xrightarrow{0} (m_2, m_1, 0, 0) \xrightarrow{3}$
$(m_2, m_1, 3, 3) \xrightarrow{0} (m_2, \star, 3, \perp) \xrightarrow{2} (m_2, \star, 5, \perp) \xrightarrow{0}$
$(\star, \star, \perp, \perp)$

s_2 :
$(\overline{m}_1, \overline{m}_1, \perp, \perp) \xrightarrow{0} (\overline{m}_1, m_1, \perp, 0) \xrightarrow{3} (\overline{m}_1, m_1, \perp, 3) \xrightarrow{0}$
$(\overline{m}_1, \star, \perp, \perp) \xrightarrow{0} (m_1, \star, 0, \perp) \xrightarrow{4} (m_1, \star, 4, \perp) \xrightarrow{0}$
$(\overline{m}_2, \star, \perp, \perp) \xrightarrow{0} (m_2, \star, 0, \perp) \xrightarrow{5} (m_2, \star, 5, \perp) \xrightarrow{0}$
$(\star, \star, \perp, \perp)$

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.

The reader probably noticed that the dynamic model used here does not fit exactly into the discrete time synchronous framework previously

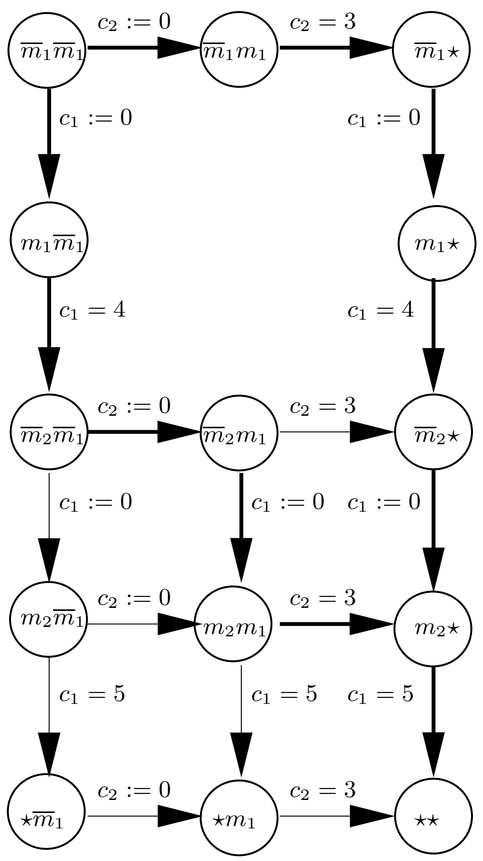

Fig. 5. The global timed automaton for the two jobs. The paths that correspond to the two schedules are indicated by thicker arrows.

described. By using a "sampled" approach and restricting events to occur and to be observed only at multiples of some constant δ, 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 it allows to "accelerate" the evolution of the system by letting time advance until the next event.

10. 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

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

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 $d = (d_1, d_2) \in [2, 4] \times [2, 8]$. Figure 6-(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. Figure 6-(b) shows the behavior of a static worst-case strategy based on instance $(8, 4)$ and one can see that is 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 Figure 6-(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.

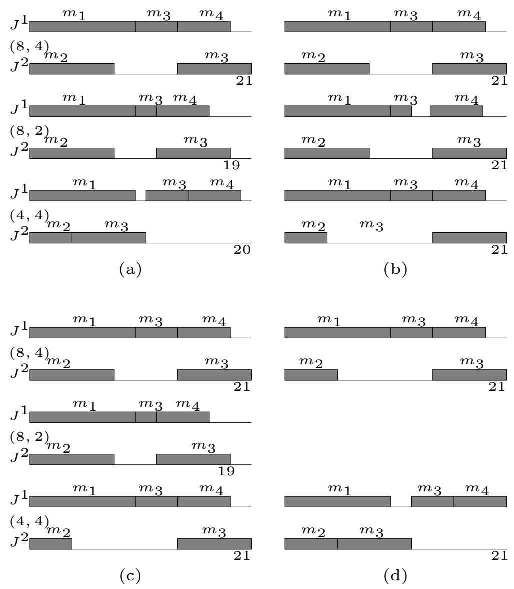

Fig. 6. (a) Optimal schedules for three instances; (b) A static schedule based on the worst instance $(8,4)$; (c) The behavior of a hole filling strategy based on instance $(8,4)$; (d) The equal performance of the two strategies on instance $(5,4)$.

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 Figure 6-(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 the schedules obtained this way 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 restricts its applicability to "slow" processes. In the next section we present an alternative approach where an equivalent strategy is synthesized *offline* using a symbolic variant of dynamic programming adapted for timed automata.

11. 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 t time pass and then execute one transition that is enabled by the clock values.* 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 the waiting time t and the value of the state reached after the transition. This is captured by the following recursive definition:

$$\begin{aligned}
&\vec{\mathcal{V}}\ (\star, c) = 0\\
&\vec{\mathcal{V}}\ (q, c) =\\
&\quad \min\{t + \vec{\mathcal{V}}\ (q', c') : (q,c) \xrightarrow{t} (q, c + t\mathbf{1}) \xrightarrow{0} (q', c')\}
\end{aligned}$$

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:

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

Figure 7 shows the final part of the global automaton corresponding to the problem, which includes state $(m_1, \overline{m}_3)$ where a decision of the scheduler has to be taken. The computation starts with $\vec{\mathcal{V}}\ (\star, \star, \bot, \bot) = 0$. The value of $(m_4, \star, c_1, \bot)$ is the time it takes to satisfy the condition $c_1 = 5$, which is $5 \dot{-} c_1$. Likewise $\vec{\mathcal{V}}\ (\star, m_3, \bot, c_2) = 7 \dot{-} c_2$. 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:

$$\vec{\mathcal{V}}\ (m_4, m_3, c_1, c_2)$$

$$= \min \left\{ \begin{array}{l} 7 \dot{-}\ c_2 + \vec{\mathcal{V}}\ (m_4, \star, c_1 + (7 \dot{-}\ c_2), \bot), \\ 5 \dot{-}\ c_1 + \vec{\mathcal{V}}\ (\star, m_3, \bot, c_2 + (5 \dot{-}\ c_1)) \end{array} \right\}$$

$$= \min\{5 \dot{-}\ c_1, 7 \dot{-}\ c_2\}$$

$$= \left\{ \begin{array}{l} 5 \dot{-}\ c_1 \text{ if } c_2 \dot{-}\ c_1 \geq 2 \\ 7 \dot{-}\ c_2 \text{ if } c_2 \dot{-}\ c_1 \leq 2 \end{array} \right.$$

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 $(m_1, \overline{m}_3)$ 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:

$$\vec{\mathcal{V}}\ (m_1, \overline{m}_3, c_1, \bot) = \min\{16, 21 \dot{-}\ c_1\}$$

$$= \left\{ \begin{array}{ll} 16 & \text{if } c_1 \leq 5 \\ 21 \dot{-}\ c_1 & \text{if } c_1 \geq 5 \end{array} \right.$$

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. Figure 6-(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 abovementioned 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 12.54%. The hole filling strategy deviates from the optimum by 4.90% (for optimistic prediction) and 4.44% (for pessimistic prediction). Our strategy produces schedules that are longer than the optimum by 1.40% and 1.14%, respectively.

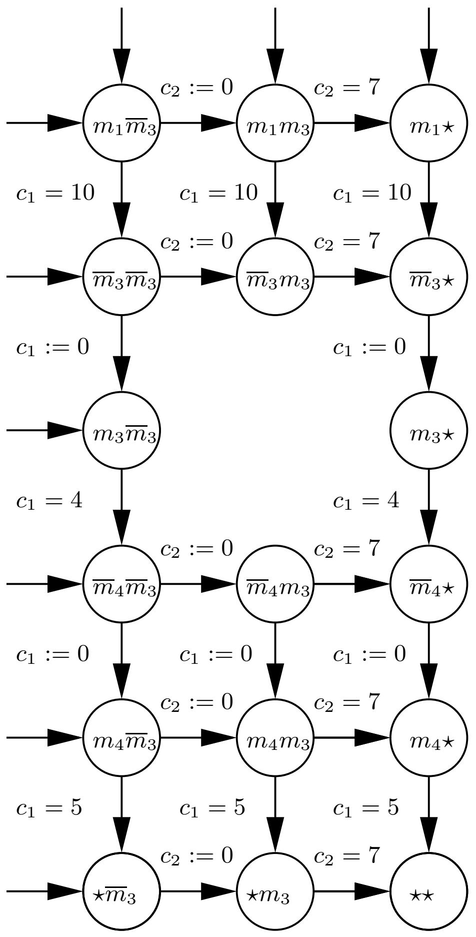

Fig. 7. Part of the global automaton.

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.

12. 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 the 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 [A02] or the paper [AAM04].

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, Anders Ravn, Tariq Samad, Michel Sintzoff and Pravin Varaiya, and from numerous past discussions with Eugene Asarin and Bruce Krogh.

References

[A02] Y. Abdeddaïm, *Scheduling with Timed Automata.* PhD thesis, Institut National Polytechnique de Grenoble, Laboratoire Verimag, 2002.

[AAM04] Y. Abdeddaïm, E. Asarin and O. Maler, Scheduling with Timed Automata, *Theoretical Computer Science*, 2004.