On Optimal and Reasonable Control in the Presence of Adversaries

Oded Maler

CNRS-VERIMAG
Grenoble, France

August 2005

What Not

- New ”results” and theorems
- Description of application or quasi-applications with tables of performance results

Results and applications are not necessarily pejorative (when done with moderation) but this is not all you need all the time

So What Then?

- A unified framework for defining system design problems using dynamic games. It covers things done under different titles by numerous communities and disciplines
- An examination of three general classes of methods for finding optimal strategies
- A sketch of my work on one instance of this scheme, the modeling and solution of some dynamic scheduling problems

The Special Theory of Everything

We want to build something (controller) that interacts with some part of the ”real” world (environment) such that the outcome of this interaction will be as good as possible

Our starting point (which is not self-evident) is that we have a mathematical model of the dynamics of the environment, including the influence of the controller’s actions

We want to use this model to choose/compute a good/optimal/satisfactory controller out of a given class of controllers

Games

The mathematical model: a two-player dynamic antagonistic game with:

- X - the (neutral) state space of the environment
- U - the set of possible actions of the controller
- V - the set of uncontrolled actions of the environment (uncertainty, disturbance, imprecise modeling, user requests..)

We want the controller to choose the best $u \in U$ in each situation, and to steer the game in the optimal direction

But what does optimal mean when the outcome is dependent also on the actions of the other player?

How to Evalute/Optimize Open Systems

Consider a one-shot game a-la von Neumann and Morgenstern

The outcome be defined as $c : U \times V \to \mathbb{R}$

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

Worst-case: $u = argmin\ \max\{c(u, v_1), c(u, v_2)\}$
Average case: $u = argmin\ p(v_1) \cdot c(u, v_1) + p(v_2) \cdot c(u, v_2)$
Typical case: $u = argmin\ c(u, v_1)$

Remark: worst-case criterion ignores performance on other cases, while average-case takes them into account

Dynamic Games

Reactive systems, ongoing interaction between controller and environment

State space X and a dynamic rule of the form $x' = f(x, u, v)$, which determines the next state as a function of the actions of the two players

In discrete time: $x_i = f(x_{i-1}, u_i, v_i)$

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

There are other more "asynchronous" games

Initial state x_0.

Runs of a Game

A sequence $\bar{u} = u[1], \ldots, u[k]$ of controller actions and
A sequence $\bar{v} = v[1], \ldots, v[k]$ of environment actions
(no matter how generated) determine a unique trajectory (run, sequence, behavior)

$$\bar{x} = x[0], x[1], \ldots, x[k] \quad \text{s.t}$$

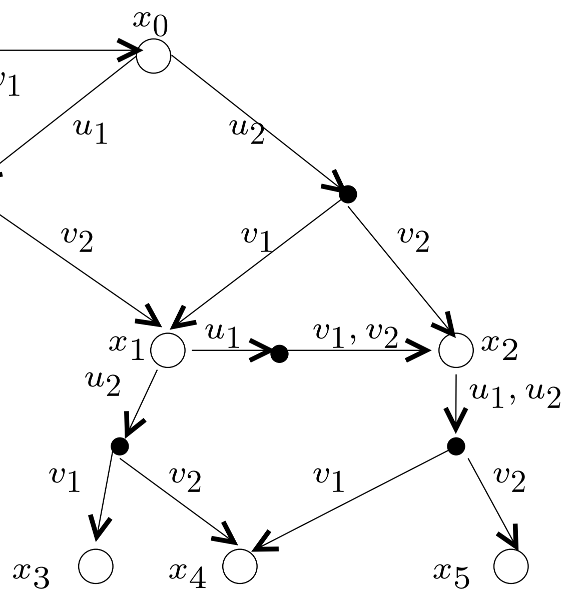
$$x[0] = x_0 \quad \text{and}$$
$$x[t] = f(x[t-1], u[t], v[t]) \ \forall t$$

We say that $\bar{x}$ is the run of the game induced by $\bar{u}$ and $\bar{v}$ and write it as the predicate/constraint $B(\bar{x}, \bar{u}, \bar{v})$ or:

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

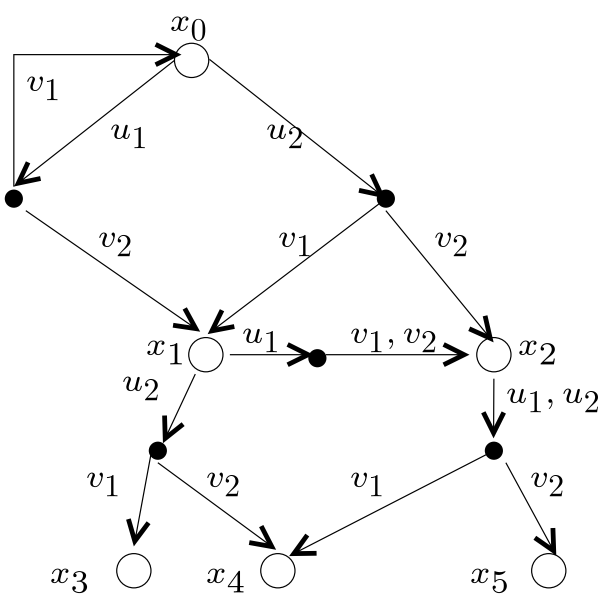
Graphically Speaking

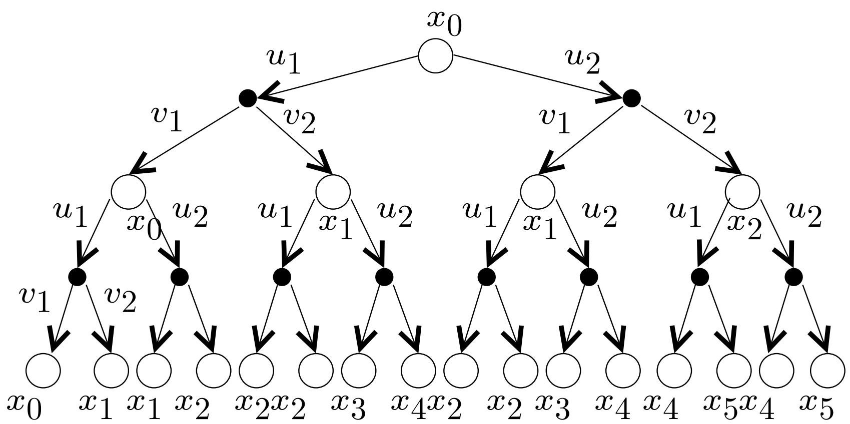
For discrete systems we can draw the game as a graph where every run corresponds to a (labeled) path



Treely Speaking

By unfolding the graph into a tree we get an enumeration of all paths





Defining Optimal Controllers

We want to choose/compute a controller/strategy/policy for choosing u which is optimal in some sense. The define the sense we need to specify:

- How to assign costs to individual runs
- What class of controllers (with/out feedback, with/out memory)
- How to evaluate over choices of the adversary (worst-case, etc.)

Assigning Costs to Trajectories

We can associate costs $c(x, u, v)$ with transitions, which reflects the ”goodness” of $x' = f(x, u, v)$, the cost of the control action u and the uncontrolled cost of v

We can then “lift” this cost to trajectories either by summation (with/out discounting):

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

(special case: minimal time/cost to reach a target set F)

or by max:

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

(special case: verification of safety properties, avoiding a bad set B)

Remark: Sub Models

Sub models of the general model are obtained by suppressing one of the players and considering it deterministic

$$\begin{tikzcd}[column sep=small]
& \begin{array}{c}(X, U, V)\\ \text{Game, strategy, synthesis}\end{array} \arrow[dl, dash] \arrow[dr, dash] & \\
\begin{array}{c}(X, U)\\ \text{Planning, open-loop}\end{array} \arrow[dr, dash] & & \begin{array}{c}(X, V)\\ \text{Verification of a given controller}\end{array} \arrow[dl, dash] \\
& \begin{array}{c}(X)\\ \text{Single trajectory, simulation}\end{array} &
\end{tikzcd}$$

Three Generic Solution Methods

- Bounded horizon and finite-dimensional constrained optimization (model-predictive control, bounded model-checking, SAT-based planning)

- Dynamic Programming (value function, Bellman-Ford, HJBI, MDPs)

- Heuristic Search (best-first, evaluation function, game-playing programs)

Bounded Horizon Problems

Comparing strategies based on behaviors of fixed length

Justifications:

1) In many problems of "control to target" and "shortest path" all desirable behaviors reach a goal state after finitely many steps

2) Looking too far in the future is anyway unreliable (model-predictive control)

3) The problem can be reduced to standard finite dimensional optimization

Bounded Horizon Problems without Adversary

For $x' = f(x, u)$ we look for a sequence $\bar{u} = u[1], \ldots, u[k]$ which is the solution of the constrained optimization problem

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

Here $c(\bar{x}, \bar{u})$ is the function defining the cost of the run $\bar{x}$ and the control actions $\bar{u}$ while $B(\bar{x}, \bar{u})$ is the constraint that $\bar{x}$ is indeed induced by $\bar{u}$ (a conjunction obtained by k-unfolding of the transition function)

For linear dynamics, $x' = Ax + Bu$, and linear cost this reduces to linear programming

In discrete planning this reduces to Boolean satisfiability. The same goes for verification (bounded model checking)

Strategy without Adversary = Plan

Without external disturbances, the choice of $\bar{u}$ completely determines $\bar{x}$

The controller "knows" what will be $x[t]$ at every t and the strategy can be viewed as a plan, a sequence of actions

$$u[1], \ldots, u[k]$$

to be taken at certain time instants without any feedback from the dynamics of the environment

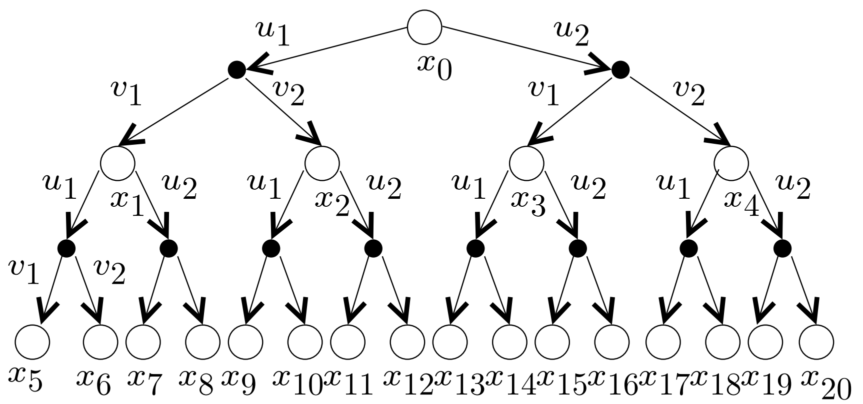
Reintroducing the Adversary

The same problem with adversary, applying the worst-case criterion, is:

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

We can enumerate all the possible control sequences and compute their cost:

$$u_1 u_1 : \quad \max\{x_5, x_6, x_9, x_{10}\}$$
$$u_1 u_2 : \quad \max\{x_7, x_8, x_{11}, x_{12}\}$$
$$\ldots$$

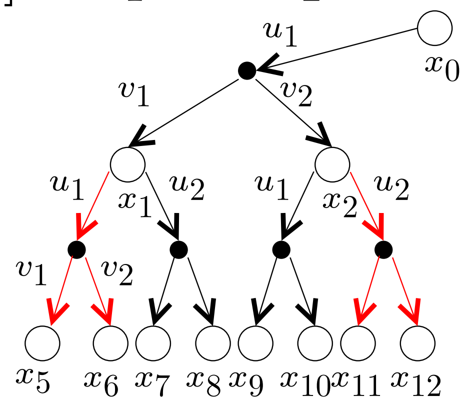


Strategies based on Feedback

The resulting sequence is the optimal "open-loop" control achievable. It ignores information obtained during execution

If $\max\{x_5, x_6\} < \max\{x_7, x_8\}$
but $\max\{x_9, x_{10}\} > \max\{x_{11}, x_{12}\}$

we should apply u_1 when $x[1] = x_1$ and u_2 when $x[1] = x_2$



Control Strategies

A (state-based) control strategy is a function $s : X \rightarrow U$ telling the controller what to do at any reachable state of the game

The following predicate indicates the fact that $\bar{x}$ is the run of the system induces by disturbance $\bar{v}$ and control $\bar{u}$ where $\bar{u}$ is computed according to strategy s:

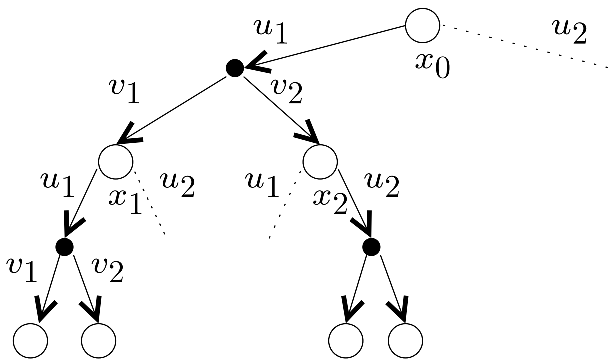
$$B_s(\bar{x}, \bar{u}, \bar{v}) \text{ iff } B(\bar{x}, \bar{u}, \bar{v}) \text{ and } u[t] = s(x[t-1]) \ \forall t$$

Finding the best strategy s is the following 2nd-order optimization problem:

$$\min_s \max_{\bar{v}} c(\bar{x}, \bar{u}, \bar{v}) \text{ subject to } B_s(\bar{x}, \bar{u}, \bar{v})$$

Computing Strategies as Restricting the Controller

A strategy removes all but on u transition in the game graph and its tree unfolding. Computing the optimal strategy is choosing the best V-induced tree



Finding an optimal strategy is typically harder than finding an optimal sequence. In discrete finite-state systems there are $|U|^{|X|}$ potential strategies and each of them induces $|V|^k$ behaviors of length k.

Worst Case is not always the Best

One weakness of the worst-case criterion is that two strategies that achieve the same performance in the worst-case but differ significantly in other cases are considered as equal

We want something stronger but which is cumbersome to express as a finite horizon optimization problem due to alternation of $\forall$ and $\exists$ ($\max$ and $\min$)

Dynamic Programming

Compute iteratively a strategy which is better than worst-case optimal

It is (worst-case) optimal from any state $x \in X$, not only from x_0

The controller does its best wherever it may find itself, not only along the worst branch

Value Function

Assume (wlog) that we evaluate trajectories according to the time/cost it takes to reach a target (and absorbing) set F

$$\sum_t c(x[t], u[t], v[t]) \qquad c(x, u, v) = 0 \text{ if } x \in F$$

A value function (cost-to-go) $\vec{\mathcal{V}}$: $X \to \mathbb{R}$ such that $\vec{\mathcal{V}}(x)$ is the best (worst-case) cost achievable by the controller from x. It is defined recursively as

$$\vec{\mathcal{V}}(x) = 0 \qquad \text{when } x \in F$$
$$\vec{\mathcal{V}}(x) = \min_u \max_v (c(x, u, v) + \vec{\mathcal{V}}(f(x, u, v)))$$

Value Iteration

Compute a monotone sequence $\vec{\mathcal{V}}_0, \vec{\mathcal{V}}_1, \ldots$ of upper-bounds for $\vec{\mathcal{V}}$ until a fixed-point is reached

$$\vec{\mathcal{V}}_0\ (x) = \begin{cases} 0 & \text{when } x \in F \\ \infty & \text{when } x \not\in F \end{cases}$$

$$\forall x \quad \vec{\mathcal{V}}_{i+1}\ (x) = \min \left\{ \begin{array}{l} \vec{\mathcal{V}}_i\ (x), \\ \min_u \max_v (c(x,u,v) + \vec{\mathcal{V}}_i\ (f(x,u,v))) \end{array} \right\}$$

Propagation backwards from F

Special Cases

Worst-case cheapest path:
$\overrightarrow{\mathcal{V}}(x) = \min_u \max_v (c(x,u,v) + \overrightarrow{\mathcal{V}}(f(x,u,v)))$

Average-case cheapest path (MDP):
$\overrightarrow{\mathcal{V}}(x) = \min_u (\sum_v p(x,v) \cdot (c(x,u,v) + \overrightarrow{\mathcal{V}}(f(x,u,v)))$

Synthesis for safety (DEDS):
$\overrightarrow{\mathcal{V}}(x) = \min_u \max_v (\max\{c(x), \overrightarrow{\mathcal{V}}(f(x,u,v))\})$

$\overrightarrow{\mathcal{V}}_i$ characterizes the states from which the controller cannot postpone reaching a forbidden state for more than i steps. Without u it is the standard backward reachability algorithm

Properties of Dynamic Programming

Guaranteed to terminate in many cases (finite graphs with non-negative costs, for example)

In continuous domains $\vec{\mathcal{V}}$ is the solution of the HJBI PDE

Derivation of strategies from value functions is straightforward (but representation in memory is less so)

Polynomial in the size of the transition graph (does NOT help us much due to curse of compositionality and dimensionality)

Major weakness: it computes $\vec{\mathcal{V}}$ over the whole state space, including states that the strategy avoids

Forward Search

The equation

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

Can be interpreted as a recursive algorithm for computing $\overrightarrow{\mathcal{V}}(x_0)$, which goes down recursively and eventually explores all the game graph and computes $\overrightarrow{\mathcal{V}}$ as does dynamic programming

A straightforward implementation is exponential in the size of the graph (due to tree unfolding) but it can be made polynomial with memorization of values

An Exhaustive Search Algorithm

```
real proc Value(x)

if x ∈ F then Val := 0
elsif x is an OR state
  Val := ∞
  forall u ∈ U do
    Val' := c(x,u) + Value(f(x,u))
    Val := min{Val, Val'}
elsif x is an AND state
  Val := 0
  forall v ∈ V do
    Val' := c(x,v) + Value(f(x,v))
    Val := max{Val, Val'}
return(Val)
```

The Advantage of Forward Search

Under certain conditions, the forward search algorithm can be transformed into an adaptive “intelligent” algorithm that attempts to focus on the interesting parts of the search space

It can find reasonable strategies while exploring only a small fraction of the game graph

This seems to be the dominant approach in AI and game playing

This is the only hope for fighting the state explosion problem

Principles of Best-first Search

To implement such a directed search you need:

Compute the cost-to-come $\overleftarrow{\mathcal{V}}(x)$ as you go down a branch

Have an easy to compute estimation function $E(x)$ which gives an approximation of $\overrightarrow{\mathcal{V}}(x)$. This is domain specific

When a state $x' = f(x, u, v)$ is a candidate for exploration, evaluate it according to $\overleftarrow{\mathcal{V}}(x) + c(x, u, v) + E(x')$

Explore the most promising branches first (plus sophisticated backtracking tricks, some randomization, anytime...)

With a proper choice of E you can sometimes find the optimal strategy without exploring the whole state space, but typically a large part needs to be explored

Giving up Exhaustiveness and Optimality

To solve really large problems we need to sacrifice optimality and avoid large parts (most) of the search space.

The effect of not exploring U branches and V branches are different

Avoiding U branches we may miss the optimal strategy and compromise on the real value of the game

Avoiding V branches we risk being too optimistic about the value of the strategy (unacceptable for safety criterion)

Avoiding V branches we may also miss some reachable states and the strategy remains incomplete - we need to augment it with some default actions in states in which it is not defined

Interim Summary

No punch line...

Variants of the same problem are attempted to be solved everywhere

The distribution of solution methods over communities is often a matter of tradition rather than adequacy

Since the algorithmic scheme is common to a variety of specific instances, maybe the principles laid down here can serve as a basis for a semi-universal synthesizer and a systematic study of the structure of game graphs for different problems

Application to Continuous and Hybrid Control

What do do when X, U and V are continuous?

One solution is to discretize U and V

Some toy examples:

Search-based verification (with J. Kapinski, B. Krogh and O. Stursberg)

Guiding a vehicle among obstacles (O. Ben Sik Ali)

Finding recovery sequences for power networks (A. Donze and S. Shapero)

Part II: Application to Scheduling

Principles:

State-space based approach

State: which tasks are waiting, enabled, executing (for how long), terminated

Controller actions: to choose which enabled tasks to start (or to wait)

Adversary actions: arrival of tasks, termination of tasks, evaluation of conditions, breaking of machines, change in criteria

Conceptual difficulty: not modeled naturally as synchronous games; more event-triggered than time triggered

Solution: modeling as timed automata = dense time + discrete transitions

Timed Systems

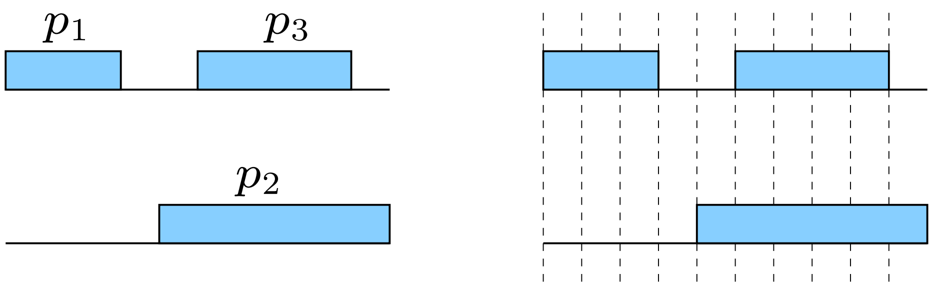
The model described so far assumes implicitly a "synchronous" time scale, where something happens every time instant

Some application domains such as scheduling, digital circuit timing analysis, real-time systems, have a more "asynchronous" nature

Typical behaviors consist of sparse events (starting, ending, rising, falling) separated by long periods where the only thing that happens is the passage of time

Timed automata are the natural dynamic model for such systems, on which controller synthesis can be done

Synchronous Modeling Style



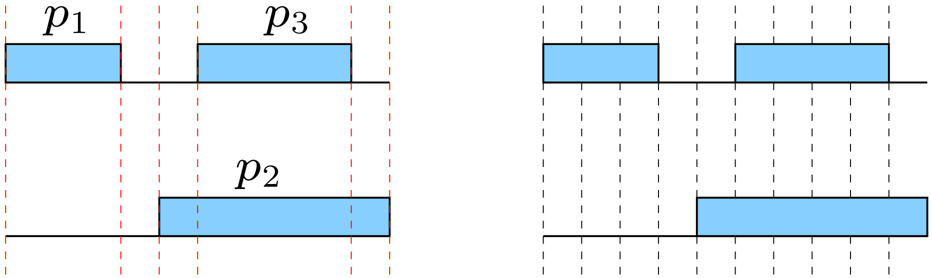
We can discretize time and have a similar type of a dynamical system where actions of the controller are $\perp$ (do nothing) and st_i (start executing p_i). The actions of the environment are $\perp$ and en_i (terminate p_i)

$$\xrightarrow{st_1} p_1 \xrightarrow{\perp} p_1 \xrightarrow{\perp} p_1 \xrightarrow{\perp, en_1} \emptyset \xrightarrow{\perp,} \emptyset \xrightarrow{\perp, st_2} p_2 \xrightarrow{\perp, st_3} \{p_2, p_3\}$$

$$\xrightarrow{\perp} \{p_2, p_3\} \xrightarrow{\perp} \{p_2, p_3\} \xrightarrow{\perp} \{p_2, p_3\} \xrightarrow{\perp, en_3} p_2 \xrightarrow{\perp, en_2} \emptyset$$

Asynchronous, Event-Triggered, Timed Style

The time index is not time but the events



$$\xrightarrow{st_1} (p_1, 0) \xrightarrow{3} (p_1, 3) \xrightarrow{en_1} \emptyset \xrightarrow{1} (p_2, 0) \xrightarrow{1} (p_2, 1) \xrightarrow{st_3} \{(p_2, 1), (p_3, 0)\}$$
$$\xrightarrow{4} \{(p_2, 5), (p_3, 4)\} \xrightarrow{en_2} (p_2, 5) \xrightarrow{1} (p_2, 6) \xrightarrow{en_2} \emptyset$$

Timed automata express processes that alternate between time passage (without a-priori commitment to a time step) and discrete transitions. Clocks measure elapsed time since transitions and are part of the state-space

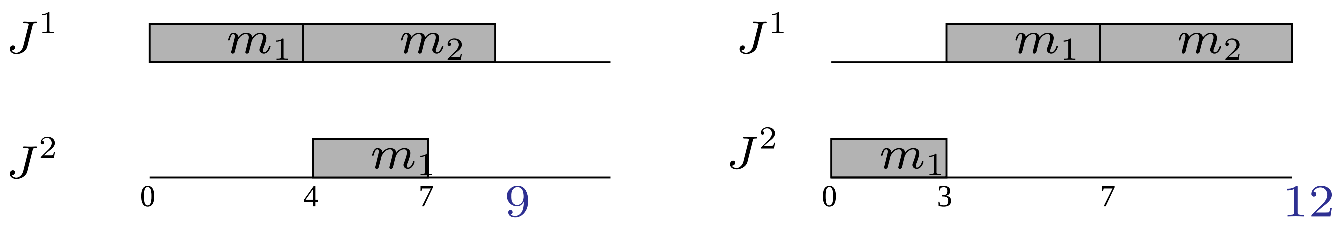
Example: Deterministic Job-Shop Scheduling

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

Determine the execution times of the steps/tasks such that:

The termination time of the last step is minimal

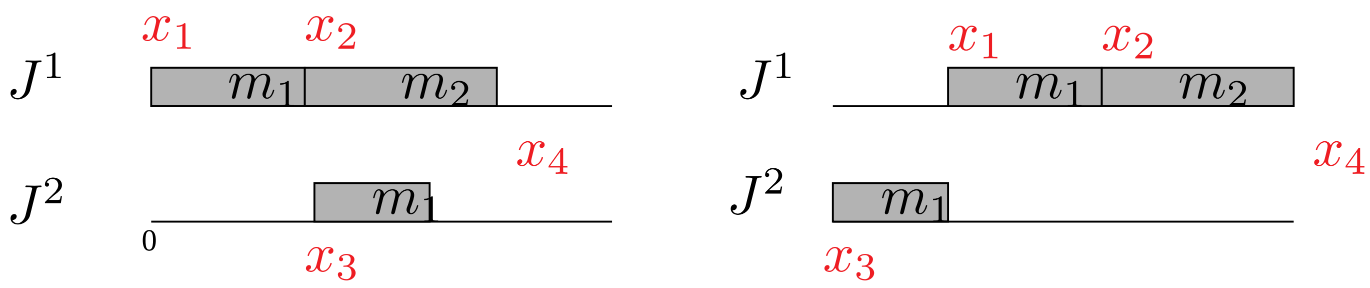
Precedence and resource constraints are satisfied



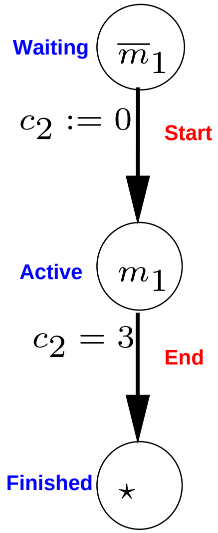
Sometimes it is better not to start a step although the machine is idle

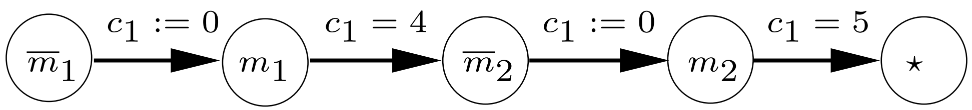
Constrained Optimization (Bounded Horizon)

minimize x_4 subject to	(makespan)	minimize x_4 subject to
$x_2 \geq x_1 + 4$ $x_4 \geq x_2 + 5$ $x_4 \geq x_3 + 3$	(precedence)	$x_2 - x_1 \geq 4$ $x_4 - x_2 \geq 5$ $x_4 - x_3 \geq 3$
$[x_1, x_1 + 4] \cap$ $[x_3, x_3 + 3]$	(mutual exclusion)	$x_3 - x_1 \geq 4 \vee$ $x_1 - x_3 \geq 3$



Modeling with Timed Automata



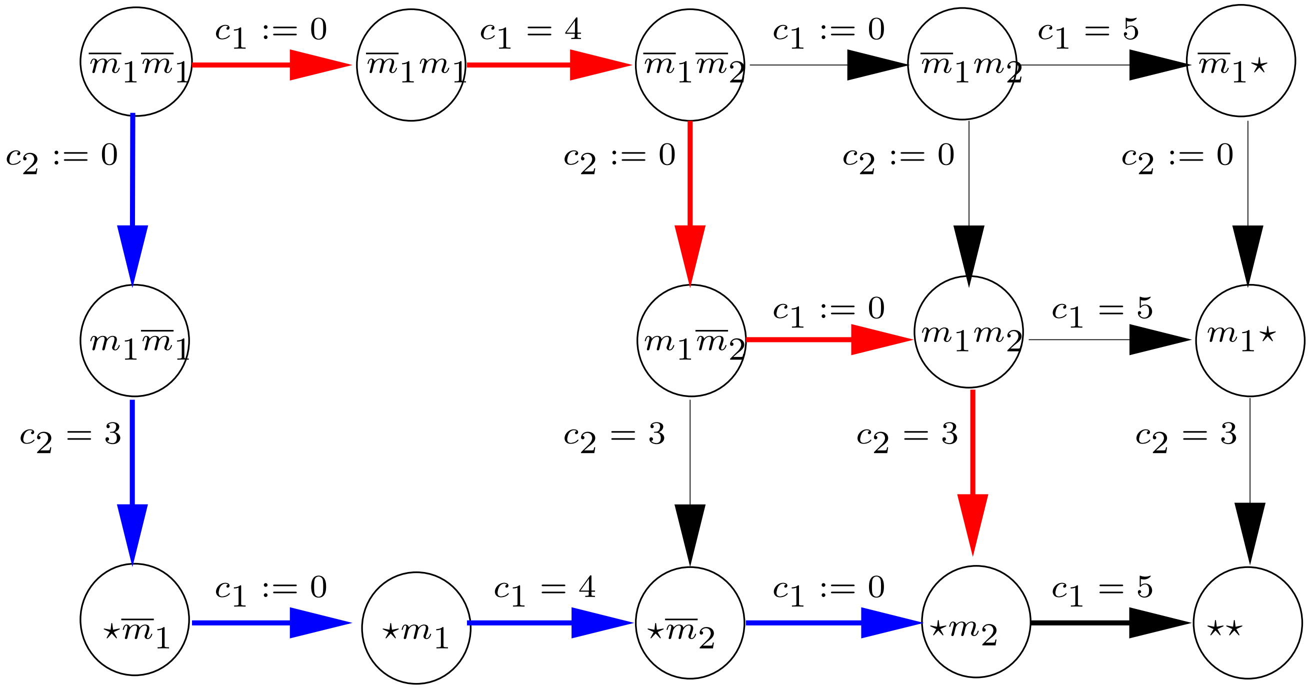


Each automaton represents the set of all possible behaviors of each task/job in isolation (respecting the precedence constraints)

The Start transitions are issued by the controller/scheduler and the End transitions by the environment

The Global Automaton

Resource constraints expressed via forbidden states in the product automaton



Optimal scheduling = shortest path problem timed automata

State-of-this-Art

Deterministic Job-Shop: search algorithms on automata (with heuristics) are not worse than other methods (with Y. Abdeddaïm, 2001)

Extension to deterministic task-graph problem. More general precedence constraints than in job-shop, uniform machines (Y. Abdeddaïm and A. Kerbaa 2003)

Extension to preemptive job-shop using stopwatch automata (Y. Abdeddaïm, 2002)

Strategy synthesis for job-shop with uncertainty in task durations. Steps of the form $(m_1, [2, 5])$. Strategy better than static worst-case (E. Asarin and Y. Abdeddaïm 2003)

Strategy synthesis for conditional precedence graph. Whether or not some tasks need to be executed will be known only after termination of other tasks (M. Bozga and A. Kerbaa)

Summary

Dynamic games are a natural model for many many problems in system design. The interesting questions about games are not necessarily those asked by “game theorists”

Clean semantic modeling precedes (but of course, does not replace) optimization algorithms

Scheduling could benefit from a general theory based on these principles