Optimization of Multi-Agent Motion Programs with Applications to Robotic Marionettes

Patrick Martin and Magnus Egerstedt

Georgia Institute of Technology, Atlanta, GA 30332, USA
{pmartin, magnus}@ece.gatech.edu

Abstract. In this paper, we consider the problem of generating optimized, executable control code from high-level, symbolic specifications. In particular, we construct symbolic control programs using strings from a motion description language with a nominal set of motion parameters, such as temporal duration and energy, embedded within each mode. These parameters are then optimized over, using tools from optimal switch-time control and decentralized optimization of separable network problems. The resulting methodology is applied to the problem of controlling robotic marionettes, and we demonstrate its operation on an example scenario involving symbolic puppet plays defined for multiple puppets.

1 Introduction

In order to manage the complexity associated with many emerging controls applications, various abstraction-based formalisms have been advanced for specifying, modeling, and controlling such systems. Examples include linear temporal logic specifications (e.g. [**?**,**?**]), Maneuver Automata for capturing symmetries [**?**,**?**], and Motion Description Languages (MDL) for symbolic control (e.g. [**?**,**?**,**?**,**?**]). These different formalisms have been designed with alternative goals in mind. As such, they have different strengths, but common to them all is that they use varying degrees of abstraction to achieve desired levels of control code granularity [**?**]. However, there is always a choice to be made when mapping these high-level programs onto executable code. This mapping is the main question under consideration in this paper. In particular, we investigate how to turn such high-level control descriptions into *optimized*, *executable* low-level control software modules, or *control code*, for a particular hardware platform.

In this paper we choose the MDL framework, as originally formulated in [**?**], to break up the control task into "strings" of individual controller-interrupt pairs. However, we use a slightly modified MDL structure for the motion programs in that they support energy parameterized motions as well as novel spatio-temporal motion constraints. In particular, this work is applied to the problem of robotic puppetry. Puppeteers script plays that designate a string of motions for each character within a structured environment; consequently, the use of MDL strings for *specifying* plays is natural as observed in [**?**]. As such we script plays using

the MDL formalism and take the resulting nominal symbolic descriptions of the play and generate optimized, executable programs based on the system dynamics and an associated cost criterion.

The resulting optimization problem is not unique to puppetry, since MDL-based abstractions of hybrid systems may need to optimize their motion programs in order to account for system dynamics and constraints in a number of other applications. We approach the solution to this problem by drawing from recent results in switched-time optimization [**?**,**?**,**?**,**?**], focusing on the scheduling of discrete transitions in a hybrid system by adjusting the timing parameters or mode order of the program.

This paper expands previous work on robotic puppetry [**?**,**?**] by fully incorporating spatial and temporal constraints into the hybrid optimization engine. We do so by applying classical results in separable programming to generate an algorithm for hybrid optimization under *networked* constraints. The result of this effort is a tool (the *MDL compiler*) that is able to accept MDL strings for a collection of puppets and generate optimal timing and energy parameters under temporal and spatial constraints. Moreover, we validate this MDL compilation framework with numerical simulations involving multiple puppets with spatial and temporal constraints.

The remainder of this paper is organized as follows: In Section **??** we introduce the MDL structure and derive an optimal control-based MDL compiler. Section **??** showcases the application of this MDL compiler by optimizing motion programs involving multiple agents and spatial constraints. Section **??** discusses an application of decentralized nonlinear programming techniques for handling motion programs with timing constraints between agents. We conclude with a brief summary in Section **??**.

2 Background

In this section we discuss the background work for generating control code from high-level specifications. Figure **??** illustrates the general flow of this control code generation process. In particular we modify the "standard" MDL formalism to enable the specification of motion programs for a collection of agents typically encountered in puppetry. Furthermore, we derive the necessary optimality conditions for the program's switch times and scaling parameters, which are then used in the *MDL Compiler* block. Finally, we illustrate the application of this MDL compiler for the special case of specifying puppet motion programs.

Fig. 1. An illustration of the process of turning high-level MDL programs into executable control code.

2.1 Motion Description Language Compiler

In order to script a motion program we describe a special MDL that accounts for four important properties of multi-agent motion programs: *who* should act, *what* motion should they do, *where* should they operate, and *when* should the action occur. We assume that the agents are identified by $i \in \mathcal{M}$, where $\mathcal{M} = \{1, \cdots, m\}$, and each agent has the dynamics,

$$\dot{x}^i = f(x^i, u^i),\ x^i \in \mathbb{R}^n,\ u^i \in \mathbb{R}^p, \tag{1}$$

where we use the superscript i to denote agent i.

We define the input to this model as one in a collection of possible feedback laws, i.e. $u^i = \kappa_j(x^i, t, \alpha_j)$, with κ_j, for some j, coming from a finite set of control laws $\mathcal{K} = \{\kappa_1, \cdots, \kappa_C\}$; additionally, α_j is an "energy"-scaling parameter that could affect speed, amplitude, or some other property of the control mode. When applying a controller of this form, we get the resulting closed-loop system dynamics $\dot{x}^i = f(x^i, t, \kappa_j(x^i, t, \alpha_j))$.

"Standard" MDL combines the controllers from $\mathcal{K}$ with a time-driven interrupt, denoted τ, that dictates the time at which the control mode interrupts, resulting in controller-interrupt pairs of the form (κ, τ). However, to allow for the specification of multi-agent programs, we add in an element for agent identification, i, and a spatially defined location, r, where the agent performs its control κ. These locations in the environment come from a set $\mathcal{R} = \{r_1, \cdots, r_l\}$. Using these additional elements, we thus define our multi-agent MDL mode as the tuple (i, κ, r, τ).

For example, if agent-i is using the two mode MDL string

$$(i, \kappa_1(\alpha_1), r_1, \tau_1)(i, \kappa_2(\alpha_2), r_1, \tau_2)$$

it must complete the motion κ_1, scaled by α_1, within region r_1 until time τ_1. (Note here that even though κ_j is a function of x^i, t, and α_j, we specify it symbolically through the α_j dependency alone.) Once this mode terminates, the second mode will execute κ_2 with scaling α_2, also in region r_1, until τ_2, which in this case signals the end of the play.

Now that we have modified MDL for composing multi-agent motion programs, we focus on developing a process for tweaking the timing and scaling parameters. For instance, an undesirable MDL mode would use a control law that potentially drives the system out of its intended region. It would be better to adjust the timing and scaling of the mode so that this is prevented. We approach this problem using calculus of variations to design a MDL compiler that accepts a nominal motion program and outputs control code based on the system dynamics, under spatio-temporal constraints.

Say we are given a single-agent (agent i) program with N modes over the time interval $[t_0, t_f]$, and we denote all switch time parameters as the vector $\bar{\tau}^i = [\tau_1^i\ \cdots\ \tau_{N-1}^i]$ and the scaling parameters as $\bar{\alpha}^i = [\alpha_1^i\ \cdots\ \alpha_N^i]$. Then the

cost functional for optimizing this agent's program could take the form,

$$\min_{\bar{\tau}^i, \bar{\alpha}^i} J(\bar{\tau}^i, \bar{\alpha}^i) = \int_{t_0}^{t_f} L(x^i, t)dt + \sum_{j=1}^{N} C_j(\alpha_j^i) + \sum_{k=1}^{N-1} \big(\Psi_k(x^i(\tau_k^i)) + \Delta_k(\tau_k^i)\big). \quad (2)$$

The interpretation here is that the agent has a trajectory cost, $L(x^i, t)$, associated with the execution of the motion program. Since scaling controller speed or amplitude requires more energy, we penalize the energy usage of each mode with the $C_j(\alpha_j^i)$ functions. We also encode the spatial constraint for each mode through the spatial cost term, $\Psi_k(x^i(\tau_k^i))$, that penalizes the distance of the agent from the location of the specified region. Finally, to prevent large deviations of a particular switch-time τ_k^i, we add the temporal cost function $\Delta_k(\tau_k^i)$.

To determine the first order necessary optimality conditions, we perturb all switch times and energy parameters as $\tau_k^i \to \tau_k^i + \varepsilon\theta_k^i$ and $\alpha_k^i \to \alpha_k^i + \varepsilon a_k^i$. In [**?**], the derivation for a two mode program was given and we generalize this without proof since it is a direct generalization of the of the derivation in [**?**]:

$$\begin{aligned}
&\frac{\partial J}{\partial \tau_k^i} = \lambda^i(\tau_k^{i-}) f_k(x^(\tau_k^i)) - \lambda^i(\tau_k^{i+}) f_{k+1}(x^i(\tau_k^i)) + \frac{\partial \Delta_k^i}{\partial \tau_i} = 0,\ k = 1, \cdots, N-1 \\
&\frac{\partial J}{\partial \alpha_k^i} = \mu^i(\tau_{k-1}^{i+}) = 0,\ k = 1, \cdots, N \qquad (3)
\end{aligned}$$

where we use the short-hand notation $f_k(x^i(t))$ to denote $f(x^i, t, \kappa_k(x^i, t, \alpha_k^i))$ and where τ_k^{i-} and τ_k^{i+} are the left and right limits, respectively. Moreover, the discontinuous costates (λ^i, μ^i) satisfy the costate dynamics,

$$\begin{aligned}
\dot{\lambda}^i &= -\frac{\partial J}{\partial x^i} - \lambda^i \frac{\partial f_k}{\partial x^i}, t \in (\tau_{k-1}, \tau_k) \\
\dot{\mu}^i &= \lambda^i \frac{\partial f_k}{\partial x^i}
\end{aligned}$$

and boundary conditions,

$$\begin{aligned}
\lambda^i(\tau_N^i) &= \frac{\partial \Psi_N}{\partial x^i}(x^i(\tau_N^i)) \\
\mu^i(\tau_N^i) &= \frac{\partial C_N}{\partial \alpha_N^i} \\
\lambda^i(\tau_k^{i-}) &= \lambda^i(\tau_k^{i+}) + \frac{\partial \Psi_k}{\partial x^i}(x^i(\tau_k^i)) \\
\mu^i(\tau_k^{i-}) &= \frac{\partial C_k}{\partial \alpha_k^i}
\end{aligned}$$

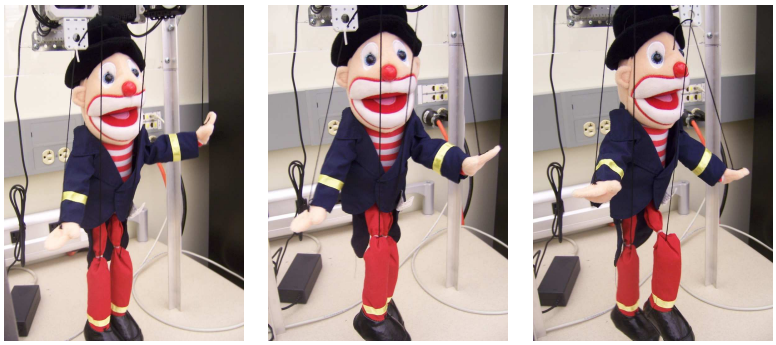
for $k = 1, \cdots, N-1$ and where we let we let $\tau_N = t_f$ and $\tau_0 = t_0$. We can now use these optimality conditions to implement a gradient descent algorithm for determining the optimized $\bar{\tau}^i$ and $\bar{\alpha}^i$ (initialized at the nominal values) as was discussed in [**?**]. This construction is in fact the fundamental tool in the MDL compilation process shown in the second block of Figure **??**.

2.2 Languages for Puppet Plays

We use the MDL compiler developed in Section **??** for coordinating multiple puppets (such as the puppet in Figure **??**), with specifications written in a special MDL for puppetry, MDLp, that mimics how puppeteers compose puppet plays. In fact, real puppet plays are written in a special script that enables the specification of puppet motion that must be choreographed with music and other puppets. Each line in a puppet play combines the agents involved, their motions, and the timing and spatial requirements [**?**,**?**]. For example, this excerpt from *Rainforest Adventures*[1] describes the motion for three puppets Female (`F`), Male 1 (`1`) and Male 2 (`2`):

```
4. F 1 2 fly up and stay and drop fast
5. F hops in place, 1 hops 4 SR turns hops 4 SL
```

The left column of the script displays the *count* number, which denotes the timing for motions for the agents listed in the line. In this case, the agents `F`, `1`, and `2` will perform several actions during the fourth count of this scene. Note that the `drop` motion is parameterized by a relative speed: `fast`. We interpret this modifier as the energy parameter α, described in Section **??**. Another important element of the play specification is the designated regions seen in count `5`: `SR` ("stage-right") and `SL` ("stage-left"). We use these stage descriptions as the regions in the set $\mathcal{R}$.



(a) (b) (c)

Fig. 2. An image sequence of the puppet executing a *wave* followed by a *walk* mode.

Accordingly, for our puppet platform, we can create several motions for the set $\mathcal{K}$ and break up the stage into the same regions used in puppetry. For example, a MDLp mode for "walking" could be written as $(1, \texttt{walk}(\alpha_1), r_2, 3)$, which is interpreted as "puppet 1 walks at speed α_1 within region 2 for 3 counts." These

[1] Courtesy of Jon Ludwig, Artistic Director of the Center for Puppetry Arts, Atlanta, Georgia, `http://www.puppet.org`.

puppetry specification language details are used when we program example plays and use the MDL compiler to generate modified control code for multi-puppet plays with spatial constraints.

3 Spatial MDL Optimization

As mentioned before, we want to specify motion programs for multiple agents, where each agent must execute the action within some region. For example, if the agent is in $\mathbb{R}^2$, with its coordinates denoted by (x, y), we could use *hard* spatial constraints to keep the x position of the agent within a set interval, i.e. $[x_1, x_2]$. However, this approach would lead to increased computational complexity as the number of agents grows, since each agent's spatial constraints have to be enforced. Alternatively, we could use *soft* spatial constraints by making the costs $L(x, t)$ and $\Psi(x(\tau))$ in (**??**) penalize (or, benefit) the spatial location of the agent. Consequently, these compiler costs are tuned depending on the particular task that each agent must achieve. In other words, given the same example agent in $\mathbb{R}^2$, we could alter the costs to completely ignore the y position, opting instead to weight only the agent's x position. Using these design considerations, we formulate an example motion program for puppets, developing cost functions for insertion into (**??**), and subsequently optimizing a play with respect to this cost functional.

3.1 Example: Spatial Optimization for Multiple Puppets

In this section we demonstrate the MDL compiler proposed in Section **??** by scripting a play with MDLp. Although the actual puppet dynamics is quite complex (e.g. [**?**]), the spatial location of the puppet can be handled without taking the joint angles into account. Instead we envision a system in which the gross spatial actuation of the puppet takes on planar unicycle dynamics. We denote each agent's planar state with the dynamics,

$$\dot{z} = \begin{bmatrix} \alpha v \sin\gamma \\ \alpha v \cos\gamma \\ u_\gamma \end{bmatrix}.$$

The α in these dynamics is the scaling parameter discussed before and v is a constant maximum speed. Additionally, γ represents the heading angle of the puppet and is driven directly by some signal u_γ. Also, the joint angles of the arms and legs are represented by the vector $q = [\theta_r\ \phi_r\ \theta_l\ \phi_l\ \psi_r\ \psi_l]^T$, where the r and l subscripts denote right and left, respectively. The motion of these arm and leg joint angles is modeled kinematically with rigid strings. Thus, the model for the joint angle motion is of the form $\dot{q} = Iu$, where I is the identity matrix and u is the chosen input signal. We concatenate the z and q states, and denote the puppet's state as $\bar{x} = [z\ q]^T$. (Note that we choose a simplified model for this puppet for less intensive algorithm computations. For a deeper examination of puppet models see [**?**,**?**].)

In this example, we want the puppets to stay as close to the center of their designated regions as possible; therefore, we use a quadratic cost for $L(\bar{x}, t)$ and $\Psi(\bar{x}(\tau))$ in (**??**). Additionally, the desired trajectory terms in these costs, denoted by $\bar{x}^d$, will depend on the regions specified in the MDLp script.

Using these cost design choices, we let $L(\bar{x}, t) = (\bar{x} - \bar{x}^d)^T P(\bar{x} - \bar{x}^d)$, where P is a 9×9 positive definite weight matrix. The other cost function that accounts for spatial penalties is $\Psi(\bar{x})$. We define this function as,

$$\Psi_k(x(\tau_k)) = (\bar{x}(\tau_k) - \bar{x}^d(\tau_k))^T Z(\bar{x}(\tau_k) - \bar{x}^d(\tau_k)), \;\; k = 1, \cdots, N-1,$$

where $Z \succ 0$ is another weight matrix. This function is similar to $L(\bar{x}, t)$; however, its weight matrix penalizes *only* the position of the agent, and it is evaluated only at the switch times, τ_k. Finally, we penalize the scaling factors and time deviations in the same way as in [**?**]: $C_j = \rho_j \alpha_j^2$, for $j = 1, \cdots, N$, and $\Delta_k(\tau_k) = w_k(T_k - \tau_k)^2$ for $k = 1, \cdots, N-1$.

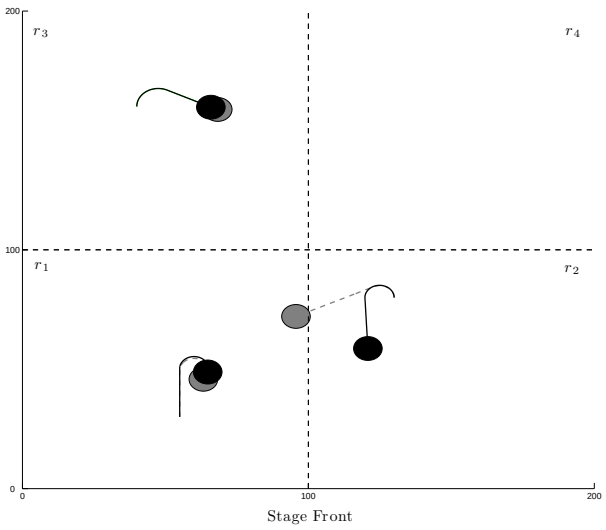


Fig. 3. Image of the puppet motions before (gray) and after (black) the MDL compilation process.

As an example, we implemented a small collection of controls, $\mathcal{K} = \{\kappa_1 = \texttt{waveLeft}, \kappa_2 = \texttt{walk}, \kappa_3 = \texttt{walkInCircles}\}$. Using these controllers, we constructed the following MDLp play:

$$(p^1, \kappa_1(1.2), r_1, 2.5)(p^1, \kappa_2(1.3), r_1, 3)(p^1, \kappa_3(1), r_1, 4)$$
$$(p^2, \kappa_1(1.2), r_3, 2.5)(p^2, \kappa_3(1.5), r_3, 2)(p^2, \kappa_2(1.3), r_3, 3)$$
$$(p^3, \kappa_3(1), r_2, 2)(p^3, \kappa_2(1.5), r_2, 4).$$

The initial run of this MDLp play is illustrated by the *gray* lines and shapes in Figure **??**. Note that puppets 1 and 2 (located in r_1 and r_3 in the figure) behave relatively well under their nominal plays. However, puppet 3 breaches the boundary between r_1 and r_2 while its MDLp string requires it to remain in r_2.

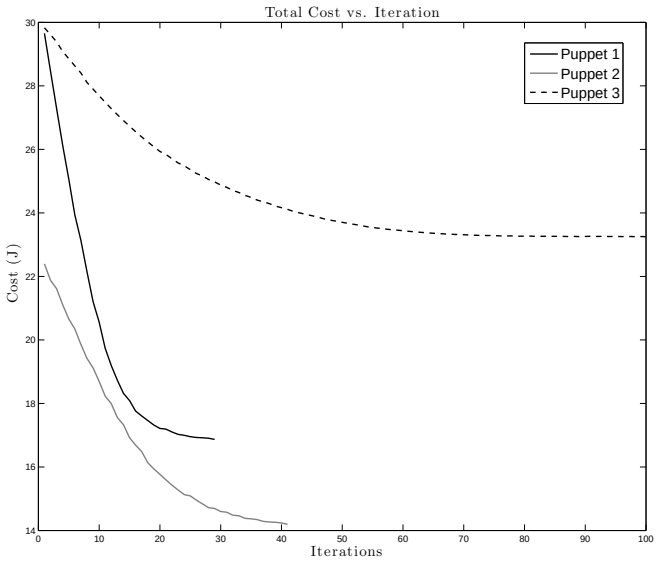


Fig. 4. This figure shows the costs as a function of the MDL compiler algorithm iteration when compiling a play for three puppets with *spatial* constraints. Puppet 1 completed in 29 iterations, Puppet 2 completed in 41 iterations, and Puppet 3 took 100 iterations.

After running the MDL compiler on these strings, the improved runtime behavior is illustrated by the black lines and shapes in Figure **??**. Puppet 3's trajectory is now correctly within r_2, as prescribed in the original MDLp string. Also, all three puppets reduce their cost, as shown in Figure **??**. Note that puppet 3 takes the longest, computing 100 iterations before minimizing its cost. This iteration count shows how bad the nominal program was at satisfying the cost functional (**??**). Additionally, our algorithm uses a conservative, fixed-step gradient descent to limit the amount of numerical error, which will slow down convergence as the derivatives (**??**) get closer to 0. If a dynamic step size were used (such as Armijo step-size [**?**]) then convergence would be faster. This work demonstrates that we can solve the problem of improving the multi-agent motion program given spatial costs. We now turn to the problem of generating optimized control code under networked timing constraints.

4 Constrained Timing Optimization

The work in Section **??** dealt with optimizing the MDL strings of multiple agents, considering each agent's dynamics and spatial costs. Additionally, many systems require *hard* timing constraints, such as terminating one particular mode *before* some other agent's mode completes. In a puppet play, missing these timing constraints may lead to benign issues, such as awkward character placement, to serious problems, such as collisions and string tangling. This section considers generating optimized MDL programs under timing inequality constraints by distributing the constraint among the agents.

In this problem, we again assume that the motion program has m puppets, each operating under their own dynamics. Additionally, each puppet switches between C_i control modes, with the terminal time denoted by $t_f = \tau_{C_i},\ i \in \mathcal{M}$. In other words, a direct modification to the formulation described in Section **??** gives that each puppet be governed by the dynamics,

$$\dot{x}^i(t) = \begin{cases} f_1(x^i(t)),\ t \in [0, \tau_1^i) \\ f_2(x^i(t)),\ t \in [\tau_1^i, \tau_2^i) \\ \vdots \\ f_{C_i}(x^i(t)),\ t \in [\tau_{C_{i-1}}^i, \tau_{C_i}^i] \end{cases}$$

for agents $i = 1, \cdots, m$. Let moreover the cost functional be defined as

$$J(\bar{\tau}^1, \ldots, \bar{\tau}^m) = \int_0^T \sum_{i=1}^m D^i(x^i, t)\ dt = \sum_{i=1}^m J^i(\bar{\tau}^i) \tag{4}$$

where $D^i(x^i, t)$ is the cost associated with operating system i for a particular control mode's time duration, without taking the other systems into account. To illustrate the way in which the temporal constraints show up, we assume, without loss of generality, that the temporal constraint only affects the d^{th} switch for systems j and k, where $j, k \in \mathcal{M}$, as $\tau_d^j - \tau_d^k \leqslant 0$. Note that this minimization formulation results in a *separable* optimization problem, since the function to be minimized (**??**) and the timing constraint depend additively on their domains [**?**].

This optimization problem can be solved by augmenting the cost with a Lagrangian term $\nu(\tau_d^j - \tau_d^k)$, and then jointly solving it across all the switching times for all the puppets. However, we do not want to use this centralized solution, since the ultimate goal is to have several autonomous agents (or in this case, puppets) optimize their plays in a decentralized fashion. Since we have already noted that the problem is separable we can break up the solution process. We specifically choose the approach known as *team theory*, recently explored in [**?**]. (Note that the details given below are not due to us, but rather that we highlight their application to the problem of distributed timing control as it pertains to the robotic marionette application.)

4.1 Distributed Timing Coordination

Puppets j and k ($j \neq k$) are temporally constrained via the d^{th} switch as $\tau_d^j - \tau_d^k \leq 0$. The constrained problem becomes

$$L(\tau_d^j, \tau_d^k, \nu) = J^j(\tau_d^j) + J^k(\tau_d^k) + \nu(\tau_d^j - \tau_d^k) \tag{5}$$

where we have assumed, without loss of generality, that the only control parameters are τ_d^j and τ_d^k, and the other switching times are considered to be fixed. It should be noted that the cost functionals are decoupled (i.e. cost J^j depends *only* on system j's dynamics). Therefore, taking the derivative of the Lagrangian with respect to ν results in the expression,

$$\frac{\partial L}{\partial \nu} = \tau_d^j - \tau_d^k,$$

in combination with the previously defined gradient expressions for the switching times.

Now, algorithmically, this formulation is interesting in that the dual problem becomes $g^\star = \max_\nu g(\nu),\ \nu \geqslant 0$, where

$$g(\nu) = \min_{\tau_d^j, \tau_d^k} \{J^j(\tau_d^j) + J^k(\tau_d^k) + \nu(\tau_d^j - \tau_d^k)\}. \tag{6}$$

Note that we are looking for a *local* minimum, since a global minimum may actually lead to behavior that is undesired by the original motion program specification.

An inefficient solution to this max-min problem would apply gradient descent on the minimization problem (**??**) and follow with a *single* gradient ascent step for $\max_\nu g(\nu)$, and repeat until a solution is found. However, since we already know that this max-min problem is separable, we can solve it using a saddle-point search algorithm, known as *Uzawa's algorithm* [**?**]. The Uzawa algorithm allows the descent *and* the ascent steps to be performed simultaneously. Consequently, we use a gradient descent for the switch times, and a gradient ascent for the Lagrange multiplier ν allowing us to decouple the numerical solution process and let the networking aspect be reflected only through the update of the multiplier, as was done in [**?**]. In fact, if we let

$$\begin{aligned}
\dot{\tau}_d^j &= -\tfrac{\partial J^j}{\partial \tau_d^j} - \nu \\
\dot{\tau}_d^k &= -\tfrac{\partial J^k}{\partial \tau_d^k} + \nu \\
\dot{\nu} &= \tau_d^j - \tau_d^k
\end{aligned}$$

all that needs to be propagated between the two systems is the value of the Lagrange multiplier ν. This observation in [**?**] leads us to a general architecture for solving networked, switched-time optimization problems, as shown in Figure **??**.

This numerical architecture lets us optimize the switch times individually at each algorithm iteration, denoted by the index i. First, we initialize both

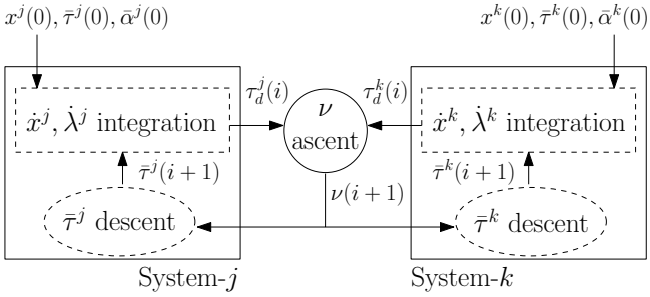


Fig. 5. This figure shows how information propagates between the two subsystems (puppets) in order to solve the networked timing problem. The initial values for system j are denoted $\bar{\tau}^j(0)$, $\bar{\alpha}^j(0)$ and similarly for system k.

systems with feasible switch times and scaling parameters based on a play of length p. These values we denote with the arrays, $\bar{\tau}^j(0) = [\tau_1^j(0) \ \cdots \ \tau_{N-1}^j(0)]$ and $\bar{\alpha} = [\alpha_1^j(0) \ \cdots \ \alpha_N^j(0)]$, respectively. Then we perform the forward-backward integration of the x^j and x^k systems and their associated co-states (λ^j, λ^k). In parallel to those integrations, the ν state is incremented using the current values for the switch times. These values are then passed to the individual systems so that they can take their gradient descent steps with the new ν values.

4.2 Example: Time-switch Constraints for Puppetry

Using the same collection of control laws from Section **??** we define a multi-puppet play as follows,

$$(1, \kappa_1(1.2), r_1, 2.5) \ (1, \kappa_2(1.3), r_1, 3) \ (1, \kappa_3(1), r_1, 3)$$
$$(2, \kappa_1(1.2), r_3, 2.5) \ (2, \kappa_3(1.5), r_3, 3) \ (2, \kappa_2(1.3), r_3, 2.5).$$

This play uses two agents, both executing three modes with various timing requirements and scaling parameters.

Following the discussion of switch-time constraints in Section **??** we choose to constrain the *first* switch of each puppet, i.e. $d = 1$. If we denote $\bar{\tau}^i = [\tau_1^i \ \tau_2^i]$ as the switch times and $\bar{\alpha}^i = [\alpha_1^i \ \alpha_2^i \ \alpha_3^i]$ as the scaling parameters for puppet i, then the constrained minimization problem for these two puppets is stated as

$$\min_{\bar{\tau}^1, \bar{\tau}^2, \bar{\alpha}^1, \bar{\alpha}^2} J^1(\bar{\tau}^1, \bar{\alpha}^1) + J^2(\bar{\tau}^2, \bar{\alpha}^2)$$
$$\text{s.t. } \tau_1^2 \leqslant \tau_1^1.$$

We formulate a Lagrangian for this problem in a similar way as equation (**??**),

$$L(\bar{\tau}^1, \bar{\alpha}^1, \bar{\tau}^2, \bar{\alpha}^2, \nu) = J^1(\bar{\tau}^1, \bar{\alpha}^1) + J^2(\bar{\tau}^2, \bar{\alpha}^2) + \nu(\tau_1^2 - \tau_1^1),$$

and then apply the proposed algorithm approach visualized in Figure **??**.

Figure **??** displays the cost graphs for the two puppets after the execution of the distributed algorithm. The cost is indeed reduced for both puppets and, furthermore, Figure **??** shows that the required inequality constraint is satisfied. The optimal switch times and scaling parameters for puppet 1 are $\bar{\tau}^1 = [2.9906\ 3.0463]$ and $\bar{\alpha}^1 = [1.1903\ 1.3249\ 1.0204]$, respectively. Additionally, the results for puppet 2's parameters are $\bar{\tau}^2 = [2.9683\ 2.9157]$ and $\bar{\alpha}^2 = [1.1901\ 1.5228\ 1.3124]$.

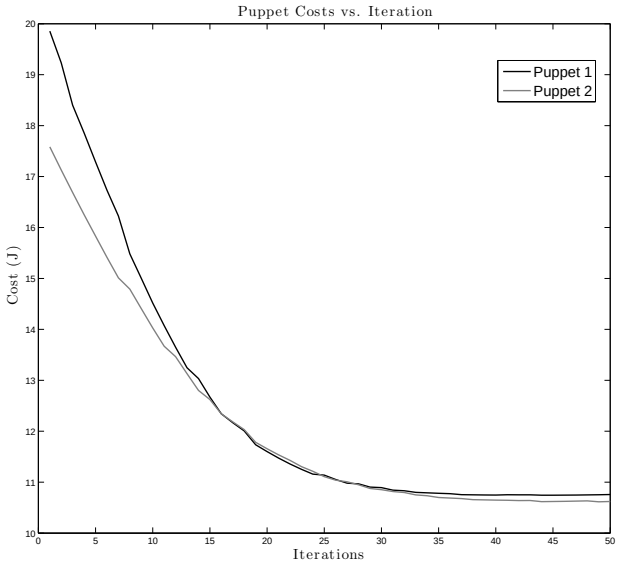


Fig. 6. The cost of both puppets using the distributed switch time constraint architecture.

5 Conclusion

In this paper we discussed recent results for generating optimized control code for collections of interacting MDL-based systems, which are, in this case, robotic puppets. We formulated a special instantiation of the MDL framework that includes spatial costs and controller energy scaling. An optimal control-based compiler was developed for these types of MDLs, and applied to a collection of autonomous puppets. Finally, our work concludes with an examination and simulation of optimizing the MDL motion program for agents with timing constraints.

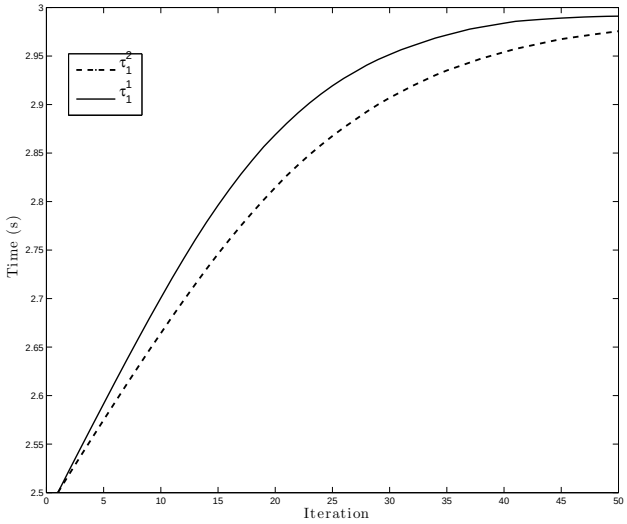


Fig. 7. A graph of the constrained switch time values τ_1^2 and τ_1^1.

Acknowledgement

We acknowledge the U.S. National Science Foundation for its support through grant number 0757317. Additionally, we thank our collaborators Todd Murphey at Northwestern University and Jon Ludwig at the Atlanta Center for the Puppetry Arts.

References

1. L. Armijo. Minimization of Functions Having Lipschitz Continuous First-Partial Derivatives. *Pacific Journal of Mathematics*, Vol. 16, pp. 1-3, 1966.
2. K. Arrow, L. Hurwicz, and H. Uzawa, *Studies in Nonlinear Programming*, Stanford University Press, Stanford, CA, 1958.
3. S.A. Attia, M. Alamir, and C. Canudas de Wit. Sub Optimal Control of Switched Nonlinear Systems Under Location and Switching Constraints. *Proc. 16th IFAC World Congress*, Prague, the Czech Republic, July 3-8, 2005.
4. B. Baird. *The Art of the Puppet*. Mcmillan Company, New York, 1965.
5. C. Belta, A. Bicchi, M. Egerstedt, E. Frazzoli, E. Klavins, and G. Pappas. Sybolic Planning and Control of Robot Motion. *IEEE Robotics and Automation Magazine*, March 2007.
6. R.W. Brockett. On the Computer Control of Movement. In the *Proceedings of the 1988 IEEE Conference on Robotics and Automation*, pp. 534–540, New York, April 1988.

7. M. Egerstedt and R.W. Brockett. Feedback Can Reduce the Specification Complexity of Motor Programs. *IEEE Transactions on Automatic Control*, Vol. 48, No. 2, pp. 213–223, Feb. 2003.
8. M. Egerstedt, T.Murphey, and J. Ludwig. Motion Programs for Puppet Choreography and Control. *Hybrid Systems: Computation and Control*, Springer-Verlag, pp. 190-202, Pisa, Italy April 2007.
9. M. Egerstedt, Y. Wardi, and H. Axelsson. Transition-Time Optimization for Switched-Mode Dynamical Systems. *IEEE Trans. on Automatic Control*, Vol. AC-51, pp. 110-115, 2006.
10. E. Frazzoli. Explicit Solutions for Optimal Maneuver-based Motion Planning. In *Proceedings of 42^{nd} IEEE Conference on Decision and Control*, Vol. 4, pp. 3372 - 3377, December 2003.
11. E. Frazzoli, M. A. Dahleh, and E. Feron. Maneuver-Based Motion Planning for Nonlinear Systems with Symmetries. *IEEE Transactions on Robotics*, 21(6):1077-1091, December 2005.
12. E. Johnson and T. Murphey, Dynamic Modeling and Motion Planning for Marionettes: Rigid Bodies Articulated by Massless Strings. In *International Conference on Robotics and Automation*, pp. 330 - 335, April 2007.
13. M. Kloetzer and C. Belta, A Fully Automated Framework for Control of Linear Systems From Temporal Logic Specifications, *IEEE Transactions on Automatic Control*, vol. 53, no.1, pp. 287-297, 2008
14. V. Manikonda, P. S. Krishnaprasad, and J. Hendler. Languages, behaviors, hybrid architectures and motion control. In J.C. Willems and J. Baillieul (eds.) *Mathematical Control Theory*, Springer-Verlag, 1998.
15. P. Martin and M. Egerstedt. Optimal Timing Control of Interconnected, Switched Systems with Applications to Robotic Marionettes. *Workshop on Discrete Event Systems*, Gothenburg, Sweden, May 2008.
16. P. Martin, J.P. de la Croix, and M. Egerstedt. MDLn: A Motion Description Language for Networked Systems. In *Proceedings of 47^{th} IEEE Conference on Decision and Control*, December 2008.
17. P.V. Moeseke and G. de Ghellinck. Decentralization in Separable Programming. *Econometrica*, Vol. 37, No. 1, pp. 73-78, 1969.
18. A. Rantzer. On Price Mechnanisms in Linear Quadratic Team Theory. *IEEE Conference on Decision and Control*, New Orleans, LA, Dec. 2007.
19. M.S. Shaikh and P. Caines. On Trajectory Optimization for Hybrid Systems: Theory and Algorithms for Fixed Schedules. *IEEE Conference on Decision and Control*, 2002.
20. P. Tabuada and G. J. Pappas. Linear Time Logic Control of Discrete-time Linear Systems. *IEEE Transactions on Automatic Control*, 51(12), 1862-1877, December 2006.
21. X. Xu and P.J. Antsaklis. Optimal Control of Switched Systems via Nonlinear Optimization Based on Direct Differentiations of Value Functions. *Int. J. of Control*, Vol. 75, pp. 1406-1426, 2002.
22. K. Yamane, J.K. Hodgins, and H.B. Brown: "Controlling a Marionette with Human Motion Capture Data," In *International Conference on Robotics and Automation*, Volume 3, pp. 3834 - 3841, September 2003.