A Software Tool for Hybrid Control

From Empirical Data to Control Programs

BY FLORENT DELMOTTE, TEJAS R. MEHTA, AND MAGNUS EGERSTEDT

When humans instruct each other on how to solve complex navigation tasks, they typically use statements like "Do A until B, then go toward C until you see D," and so on. Such statements contain only high-level descriptions of the task at hand, whereas lower-level issues concerning the exact path to follow or what muscle groups to use are implicitly assumed. This situation is in contrast with the classic control theoretic idea of prescribing the exact inputs or actuation signals that are needed for solving the task.

In this article, we present a software tool that bridges this gap by providing a framework in which robots and other dynamic systems can be controlled using automatically generated, high-level, symbolic control programs. In particular, the automated tools extract high-level control programs from observed behaviors (possibly biological) and then produce symbolic control laws that can be executed on mobile robots to mimic the observed behavior (Figure 1).

The main question investigated here can be summarized as follows: given the assumptions about what features and measurements are relevant to the original system, can we produce hybrid control strategies that mimic the observed behavior? The resulting hybrid strategies would, for example, allow us to generate control laws for teams of mobile robots that behave similarly to groups of ants or schooling fish. In other words, can we produce multimodal control strategies in an automated fashion from observed empirical data? These empirical data can be generated by nature (as in the group of ants) or from human-operated robots. From the standpoint of naturally occurring data, the short-term aim of such a research agenda would be to learn from nature, but a more lofty, long-term goal would be to understand naturally occurring control mechanisms based on hybrid control theory. From the human-operator standpoint, the goal would be to learn effective control strategies from examples.

Similar ideas have been pursued in [3] and [10], which are based on presegmented data or predefined collections of potential control laws. Alternative approaches can also be found in literature on motion captioning [17] or in the hybrid systems identification area [12]. However, these research programs are focused mainly on fitting piecewise linear, autonomous systems to the data. In this article, we take a different view, wherein the dynamics are given and the problem is to find control laws, together with the conditions for transitions between them, defined with respect to the system dynamics. At this point, it should be noted that some of the technical results presented here have appeared in [6] and [7], but in this article, we combine these results with new work and unify them into a tool for automating the process of extracting executable control strategies from empirical data. The tool is mode optimization and data extraction (MODEbox), which is available as a MATLAB toolbox [20]. The graphical user interface for MODEbox is depicted in Figure 2. MODEbox consists of four major modules: preprocessing, motion description language (MDL) capturing, MDL to automata, and simulation. Each of these modules and their functionality is discussed in this article, and further information is available on the MODEbox Web site [20].

The outline of this article is as follows: In the section on motion description languages, a brief introduction to MDLs is given. In the next section, the MODEbox and its basic

Digital Object Identifier 10.1109/MRA.2008.919022

1070-9932/08/$25.00©2008 IEEE

functionality is introduced. This is followed by a detailed explanation of the key modules that comprise MODEbox. In particular, the section on recovering MDL strings from data details how to recover MDL strings from data. The next two sections introduce a method for reducing the size of the recovered mode set to lower complexity and show how to construct a finite automaton that reproduces the recovered MDL strings. Finally, some examples are presented in the section on robots and ants.

Motion Description Languages

The main idea behind multimodal control is to define the different modes of operation, e.g., with respect to a particular task, operating point, or data source. These modes are then combined according to some discrete switching logic, and one attempt to formalize this notion is through the concept of an MDL [5], [8], [14].

Each string (or word) in an MDL corresponds to a control program that can be operated by the control system. Slightly different versions of MDLs have been proposed, but they all share the common feature that the individual atoms (or letters), concatenated together to form the control program, can be characterized by control-interrupt pairs. In other words, given a dynamic system

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

together with a control program $(k_1, \xi_1), \ldots, (k_z, \xi_z)$, where $k_i : \mathbb{R}^p \to \mathbb{R}^k$ and $\xi_i : \mathbb{R}^p \to \{0,1\}$, the system operates on this program as $\dot{x} = f(x, k_1(h(x)))$ until $\xi_1(h(x)) = 1$. At this point, the next pair is read, and $\dot{x} = f(x, k_2(h(x)))$ until $\xi_2(h(x)) = 1$, and so on. Loosely speaking, the system evolves under triggers, i.e., it is controlled by the feedback law $k_i(y)$ until interrupt $\xi_i(y)$ goes from 0 to 1, at which point the next feedback law $k_{i+1}(y)$ is used. Note that the interrupts can also be time-triggered, but this can be incorporated by a simple augmentation of the state space.

We first assume that the input-output (I/O) spaces ($\mathbb{U}$ and $\mathbb{Y}$, respectively) in (1) are finite, which can be obtained through a quantization function. This assumption can be justified by the fact that all physical sensors and actuators have finite range and resolution. Under this assumption, the set of all possible modes $\Sigma_{\text{total}} = \mathbb{U}^{\mathbb{Y}} \times \{0,1\}^{\mathbb{Y}}$ is finite as well. Moreover, we can assume that a data point is measured only when the output or input changes value. This corresponds to the so-called Lebesgue sampling, in the sense of [2], which allows us to study only the I/O strings of finite length (given that the data were generated over a finite time horizon). Under this sampling policy, we can define a mapping $\delta : \mathbb{R}^n \times \mathbb{U} \to \mathbb{R}^n$ as $x_{q+1} = \delta(x_q, k(h(x_q)))$, given the control law $k : \mathbb{Y} \to \mathbb{U}$, with a new time update occurring whenever a new I/O value is encountered. For such a system, given the input string $(k_1, \xi_1), \ldots, (k_z, \xi_z) \in \Sigma^*$, where Σ^* is the free-monoid over a particular mode set $\Sigma \subseteq \Sigma_{\text{total}}$, the evolution of x is given by

$$\begin{cases} x_{q+1} = \delta(x_q, k_{l_q}(y_q)), \quad y_q = h(x_q) \\ l_{q+1} = l_q + \xi_{l_q}(y_q), \end{cases} \tag{2}$$

where $l_q \in \{1, \ldots, z\}$ is the position of the active mode within the input string at time q. (As an example, consider the situation in Figure 3, where a mobile robot is executing an MDL string consisting of alternating avoid-obstacle and go-to-goal modes.)

One of the objectives of MODEbox is to recover such strings of feedback-interrupt pairs from observed data, which will be discussed in more detail later.

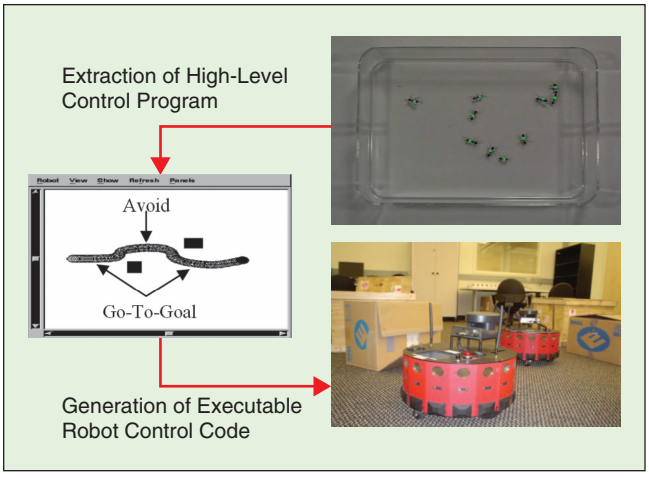

Figure 1. *An example is shown in which ten roaming ants in a tank are tracked. Their behavior is analyzed and high-level control programs are extracted for controlling teams of mobile robots.*

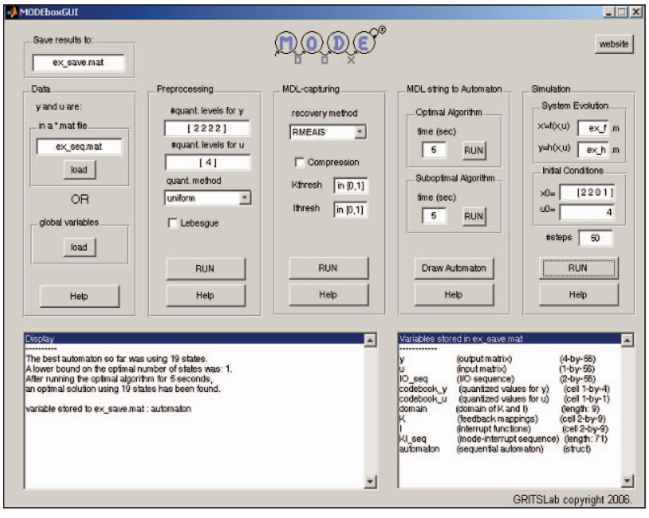

Figure 2. *Graphical user interface for MODEbox [20].*

MODEbox Basics

The basic functionality of MODEbox consists of four major modules (Figures 2 and 4). We will briefly describe each of these building blocks below and then present the theory behind their operation in later sections. Overall, MODEbox takes in a string of observed data (I/O pairs) and produces MDL strings consistent with the observed data. Next, MODEbox constructs a finite automaton capable of producing these MDL strings as a sample

path, which can then be used as a control law to simulate similar trajectories or to control real systems.

The description of each module will be accompanied by a simple maze example to better illustrate the MODEbox operation. Note that this example is overly simplistic, but it is merely to be thought of as a vehicle for making certain operational aspects explicit. Let us assume that data are collected from the observations of a robot going through a maze as depicted in Figure 5(a). These data will be given by an I/O string, where the inputs are variables relevant to the system's control decisions and the outputs are signals possibly used to control the observed system. For this particular maze example, we choose the output to be $y = (y^1, y^2, y^3, y^4)$, where y^1, y^2, y^3, and y^4 correspond to the colors (i.e., 0 is black and 1 white) of the cells in front, to the left, behind, and to the right of the robot, respectively. The input u is the corresponding action (1 stands for go straight; 2, turn left; 3, U-turn; or 4, turn right) taken by the robot in response to the outputs. These data are sent to the preprocessing block.

Block 1: Preprocessing

In the first block in Figure 4, a data string consisting of I/O pairs is being read by MODEbox. The assumption is that the data are generated by a dynamic system $\mathbf{x}_{q+1} = f(\mathbf{x}_q, \mathbf{u}_q)$, $\mathbf{y}_q = h(\mathbf{x}_q)$, and the data string is given by $(\mathbf{y}_1, \mathbf{u}_1), \ldots, (\mathbf{y}_N, \mathbf{u}_N)$, where the outputs $\mathbf{y}_i \in \mathbb{R}^p$ and the inputs $\mathbf{u}_i \in \mathbb{R}^k$. Here, we use boldface to denote variables before they have been operated on by the preprocessing block. In fact, this string is operated on by the preprocessing block using three different, sequential components, namely, `Quantize`, `Encode`, and `Lebesgue`. `Quantize` produces a finite precision representation of the data string, `Encode` maps the quantized data strings to symbols, and `Lebesgue` reduces the length of the data string by making sure that no consecutive, symbolic I/O pairs are the same [2]. As a result, the output of this block is a new string $(y_1, u_1), \ldots, (y_r, u_r)$, where $r \leq N$ and $y_i \in \mathbb{Y}$, $u_i \in \mathbb{U}$. The user can specify how many regions (`quant.numbers`) the quantization should produce and what quantization method to use (`quant.method`). The user can select between four quantization methods: uniform, equidistributed, optimal pulse code modulation (PCM), and optimal differential pulse code modulation (DPCM). The choice of a quantization method

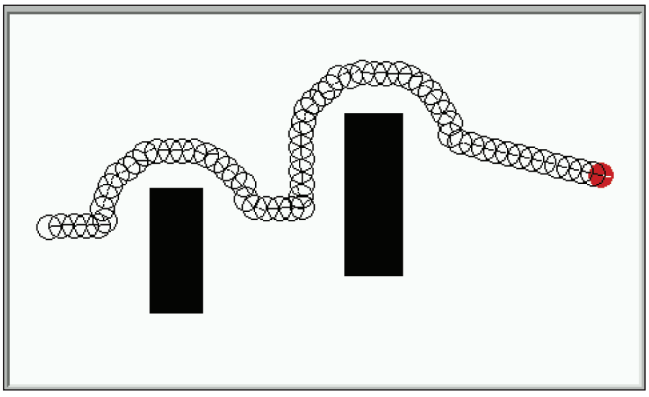

Figure 3. *A multimodal input string is used for negotiating two rectangular obstacles.*

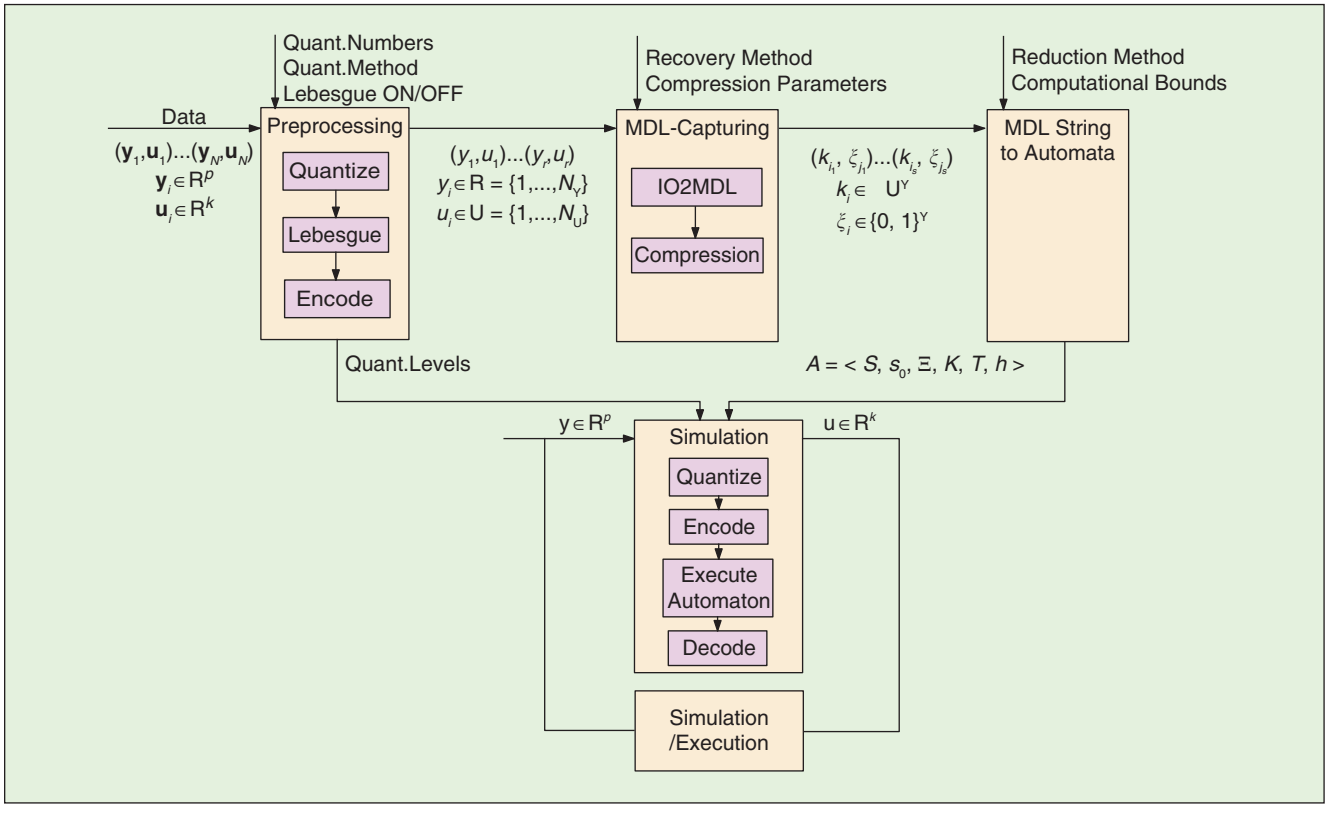

Figure 4. *Overview of the MODEbox operational units.*

The main idea behind multimodal control is to define different modes of operation in different situations.

and quantization levels is motivated by which one of these two opposite entities is to be preferred: the final model's complexity or the model's performance at approximating the original system. The user can also choose whether or not Lebesgue sampling should be employed (`Lebesgue ON/OFF`).

For the maze example, each data point comes from a small discrete set $Y \times U$, where $Y = \{0, 1\}^4$ and $U = \{1, 2, 3, 4\}$. Generally, the data could belong to countably or uncountably infinite sets. Here, data already belong to a small discrete set; therefore, we do not need to quantize the data any further. These data are now encoded into a discrete set of symbols $\mathbb{Y} \times \mathbb{U}$, where $\mathbb{Y} = \{1, 2, \ldots, 16\}$ and $\mathbb{U} = \{1, 2, 3, 4\}$, and the resulting I/O string is sent to the MDL-capturing block.

Block 2: MDL Capturing

The output from the preprocessing block is now fed into the MDL-capturing block. The method for recovering MDL strings from I/O data, `IO2MDL`, has been developed by the authors in [7], and different strategies for minimizing certain objectives have been devised. Although minimizing the so-called empirical specification complexity is the preferred objective, this is not always achievable, as noted in [7]. Instead, the user can choose from one of four methods that manage this complexity (this will be addressed in more detail in the next section). Once an MDL string is produced, the result is fed to `Compression`, where similar feedback laws and interrupt functions are identified and combined, which are based on user-specified `compression parameters` that set the thresholds (between zero and one) for how similar they need to be in order to be considered the same. The similarity measure is a normalized average entropy quantifying the uncertainty in the random variable $k_i(\gamma)$ [and, respectively, $\xi_i(\gamma)$], where i can be any of the modes under consideration. The resulting output from this block is a string $(k_{i_1}, \xi_{j_1}), \ldots, (k_{i_s}, \xi_{j_s})$, where $s \leq r$.

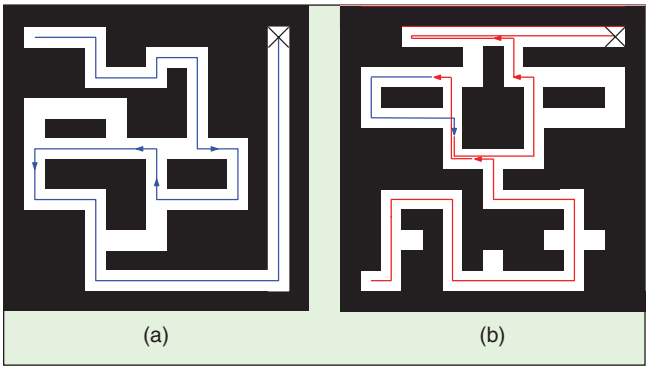

Figure 5. *(a) Trajectory of a robot going through a maze, which can be used as input data for MODEbox. The arrow and cross indicate the starting and finishing locations, respectively. (b) Example of a recovered mode sequence. Each arrow corresponds to the execution of one mode. Note that at the same moment an interrupt is triggered, a last action is taken by the active mode. For this reason, interrupts are located one step before the head of an arrow, i.e., when the robot faces a wall with two openings on its left and on its right.*

For the maze example, a close look at the trajectory in Figure 5(a) shows that the robot's behavior is almost always predictable. Indeed, the robot goes straight whenever possible (i.e., $u = 1$ when $\gamma = (1, -, -, -)^T$) and turns left or right when it is the only possible choice (i.e., $u = 2$ when $\gamma = (0, 1, 1, 0)^T$ and $u = 4$ when $\gamma = (0, 0, 1, 1)^T$). The only unpredictable situation is when the robot is facing a wall, with two openings on the left and right (situation that we will denote $\gamma_\perp = (0, 1, 1, 1)^T$). In this case, we see that the robot sometimes chose to turn left ($u = 2$) and sometimes right ($u = 4$). With this in mind, a recovery method minimizing the number of distinct modes could return the sequence $k_1\xi_1k_1\xi_1k_2\xi_2 k_1\xi_1k_1\xi_1k_1$, where mode 1 (respectively, mode 2) makes the robot turn left (right) when $\gamma_\perp$ happens (i.e., $k_1(\gamma_\perp) = 2$ and $k_2(\gamma_\perp) = 4$), where the two modes behave the same for all other situations (i.e., whenever $\gamma \neq \gamma_\perp$, $k_1(\gamma) = k_2(\gamma)$) and where the interrupts functions would be the same, only triggering when $\gamma_\perp$ happens (i.e., $\xi_1(\gamma_\perp) = \xi_2(\gamma_\perp) = 1$). This particular mode sequence is depicted in Figure 5(b). Because the two modes are almost identical, it is conceivable to merge them into a single mode (k_{12}, ξ_{12}). In this case, $k_{12}(\gamma_\perp)$ is now a random variable (and not deterministic as before) as $k_{12}(\gamma_\perp) = 2$ or 4. The resulting compressed mode sequence would be $k_{12}\xi_{12}k_{12}\xi_{12}k_{12}\xi_{12}k_{12}\xi_{12}k_{12}\xi_{12}k_{12}$. Now this mode sequence will be sent to the MDL to automata block.

Block 3: MDL to Automata

The resulting MDL string can be thought of as a sample path generated on a finite automaton, where the output function $h(s) = k$ returns the feedback law that the system should use in state s. Transitions in the automaton are triggered by the corresponding interrupts. If we let $\mathcal{K}$ and Ξ denote the set of feedback laws and interrupt functions, respectively, the MDL to automata block produces a finite automaton $A = \langle S, s_0, \Xi, \mathcal{K}, T, h \rangle$, where S is the state space, s_0 the initial state, $T : S \times \Xi \to S$ is the transition relation, and h, $\mathcal{K}$, and Ξ are as previously defined. Moreover, A should not only be such that the MDL string is a sample path of A but should also be small in the sense of state-space cardinality. This subject is considered in the "From MDL Strings to Finite Automata" section, where algorithms for finding such automata are discussed. The user is free to choose between an optimal and a suboptimal algorithm through the user input `reduction method`.

For the simple maze example, the optimal automaton is easily recovered, and the results are shown in Figure 6. An automaton corresponding to the recovered mode string without compression $(k_1\xi_1k_1\xi_1k_2\xi_2k_1\xi_1k_1\xi_1k_1)$ is shown in Figure 6(a), and an automaton corresponding to the compressed mode string $(k_{12}\xi_{12}k_{12}\xi_{12}k_{12}\xi_{12}k_{12}\xi_{12}k_{12}\xi_{12}k_{12})$ is shown in Figure 6(b).

Block 4: Simulation or Execution

Once A has been produced, it can be used as a hybrid control law to mimic observed behavior or control real systems in the simulation block. The last block in the MODEbox flow diagram represents this situation, where externally obtained measurements $\mathbf{y} \in \mathbb{R}^p$ (either through simulation or a real experiment) are quantized and encoded [with the same quantization levels (`quant.levels`) used in the preprocessing block] to produce symbolic measurements $y \in \mathbb{Y}$. These measurements are then used for driving the finite automaton through `ExecuteAutomata`, and the corresponding control symbols $u \in \mathbb{U}$ are computed and decoded to produce executable control signals $\mathbf{u} \in \mathbb{R}^k$. This is the only block in the toolbox that requires any significant user input since each simulation is application specific.

The control procedure recovered in the previous block is now applied to a robot to navigate through a new maze depicted in Figure 7. We assume that the control automaton used in this example is the one depicted in Figure 6(b). First, we define two functions $f(\mathbf{x}, \mathbf{u})$ and $h(\mathbf{x})$ reflecting the state evolution and observation of the robot traversing the maze. At each time increment, the real measurement $\mathbf{y}_q = h(\mathbf{x}_q)$ is quantized and encoded into a symbolic measurement $y_q \in \mathbb{Y}$. A symbolic input $u_q \in \mathbb{U}$ is then computed. It corresponds to the value $k_{12}(y_q)$ of the feedback mapping of the active mode. This symbolic input is decoded into an executable control $\mathbf{u} \in \mathbb{R}^k$. By applying this control to the robot, the state evolves according to $\mathbf{x}_{q+1} = f(\mathbf{x}_q, \mathbf{u}_q)$. Figure 7 shows how the simulated system exhibits trajectories or behaviors similar to those of the system used for training.

The remainder of this article is devoted to presenting the theory related to the MODEbox modules in detail and showcasing the MODEbox operation through some examples.

Recovering MDL Strings from Data

In [7], we presented different methods for recovering multimodal control strings from empirical data in a theoretical setting. In particular, the problem was to produce strings in a given MDL that were consistent with the empirical I/O strings. At the same time, the control programs were viewed as having an information-theoretic content, i.e., they could be coded more or less effectively. For this, we define a complexity measure, the empirical specification complexity, which corresponds to the number of bits needed to specify a mode string σ with an optimal coding scheme:

$$\mathcal{S}^e(\sigma) = |\sigma| \mathcal{H}^e(\sigma),$$

where $|\sigma|$ is the length of the mode sequence σ and $\mathcal{H}^e(\sigma)$ is its entropy.

The minimization of $\mathcal{S}^e(\sigma)$ is, in fact, very hard to address directly and is still an open problem. However, the easily established bound $\mathcal{S}^e(\sigma) \leq |\sigma| \log_2 (M(\sigma))$, where $M(\sigma)$ is the number of distinct modes in σ, allows us to focus on the following more tractable subproblems:

- minimizing the length of the mode sequence $|\sigma|$, which was solved using dynamic programming in [1]
- minimizing the number of distinct modes $M(\sigma)$, which was solved in [6] and relies on the initial construction of mode sequences, where the interrupts always trigger and are referred to as always interrupt sequences (AIS).

It is important to note that the solutions to these problems are not unique; hence, they can be further processed to reduce complexity. In particular, [7] presents additional algorithms to minimize entropy and reduce the length of mode strings given by the AIS solution while preserving consistency. MODEbox supports four different methods for recovering MDL strings, and the users can select among them based on their preference. The supported recovery methods are as follows: `MinLength` (minimizes the string length), `AIS` (minimizes the distinct number of modes), `RAIS` (reduces the length of a string produced by the `AIS` to further reduce the specification complexity), and `RMEAIS` (reduces the length of the lowest entropy `AIS` string). Although the `RMEAIS` produces the lowest complexity strings among `AIS`, `RAIS`, and `RMEAIS`, it does not necessarily produce strings with complexity lower than the `MinLength`.

Application of the MODEbox tool is illustrated on two different examples involving robots and ants.

Reducing the Size of the Motion Alphabet

The conversion from mode sequences to executable I/O automata requires splitting the motion alphabet Σ into two

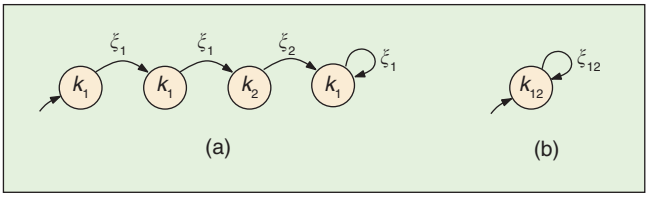

Figure 6. *(a) Automaton corresponding to the noncompressed recovered mode string. (b) Automaton corresponding to the compressed mode string.*

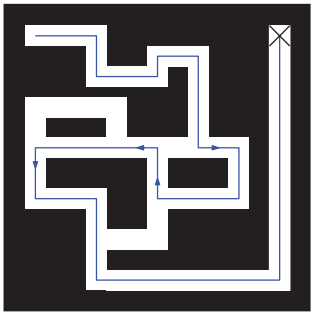

Figure 7. *Trajectory of a robot navigating through a maze using a controller derived by MODEbox.*

alphabets: the input alphabet Ξ of interrupt functions and the output alphabet $\mathcal{K}$ of feedback mappings. To produce small automata (in terms of the number of states), a preliminary task consists of reducing the size of Ξ and $\mathcal{K}$ by merging the combinations of elements that look similar. To do so, we first need to define a measure of similarity. Consider merging n distinct feedback mappings $k_{i_1}, k_{i_2}, \ldots, k_{i_n}$, resulting in the creation of a macro-feedback mapping K_I, where we let $I = \{i_1, \ldots, i_n\}$. Here, we choose to merge feedback mappings, but the same ideas, definitions, and algorithm apply for interrupt functions. Note that for a given $\gamma \in \mathbb{Y}$, two distinct feedback mappings may map γ to two different inputs, i.e., $\exists(\alpha, \beta) \in I^2$ such that $k_\alpha(\gamma) = u$ and $k_\beta(\gamma) = v$ with $u \neq v$. For this reason, we choose to represent $K_I(\gamma)$ as a random variable. Consequently, the macro-feedback mapping K_I is a random process defined on $\mathbb{Y}$.

Now, for a given $\gamma \in \mathbb{Y}$, the probability mass function of $K_I(\gamma)$ can be recovered by

$$p_{K_I(\gamma)}(u) = \Pr\{K_I(\gamma) = u\} = \frac{\text{card}\{q | \gamma_q = \gamma, u_q = u, \text{ and } m_q \in I, q = 1, \ldots, N\}}{\text{card}\{q | \gamma_q = \gamma \text{ and } m_q \in I, q = 1, \ldots, N\}},$$

where m_q refers to the active mode at time q, card denotes cardinality, and N is the number of data points in the I/O string. Next, we define the entropy of the random variable $K_I(\gamma)$:

$$H(K_I(\gamma)) = -\sum_{u \in \mathbb{U}} p_{K_I(\gamma)}(u) \log\left(p_{K_I(\gamma)}(u)\right),$$

with the following bounds

$$0 \leq H(K_I(\gamma)) \leq \log(n).$$

Finally, we define an entropy measure for the random process K_I. It is the normalized average of the entropies of all random variables $K_I(\gamma)$, $\gamma \in \mathbb{Y}$:

$$H(K_I) = \frac{1}{\log(n)} \sum_{\gamma \in \mathbb{Y}} p_Y(\gamma) H(K_I(\gamma)).$$

Note that in this definition, the output is also considered as a random variable Y. We propose three methods for establishing its probability mass function $p_Y : \gamma \to p_Y(\gamma) = \Pr\{Y = \gamma\}$:

1) γ is a uniform random variable. In this case, $p(\gamma) = 1/|\mathbb{Y}|$
2) we look at the proportion of γ when any mode in I is active, i.e., $p_Y(\gamma) = (\text{card}\{q | \gamma_q = \gamma \text{ and } m_q \in I, q = 1, \ldots, N\})/(\text{card}\{q | m_q \in I, q = 1, \ldots, N\})$
3) we look at the proportion of γ in the whole observed output sequence, i.e., $p_Y(\gamma) = (\text{card}\{q | \gamma_q = \gamma, q = 1, \ldots, N\})/N$.

The total entropy $H(K_I)$ is a measure of the average uncertainty in the random process K_I. It varies from zero to one, where

- $H(K_I) = 0$ means that there is no uncertainty in K_I, i.e., for all $\gamma \in \mathbb{Y}$, all modes in I map γ to the same input value
- $H(K_I) = 1$ means that the uncertainty in K_I is maximal, i.e., for all $\gamma \in \mathbb{Y}$, all modes are equally active, and they all map γ to a distinct input value.

In other words, the two extreme values are reached when the n feedback mappings are either equal ($H(K_I) = 0$) or completely different ($H(K_I) = 1$). We propose to define a threshold value $\gamma_k \in [0, 1]$ so that if $H(K_I) \leq \gamma_k$, the feedback mappings are considered similar enough to be merged. On the basis of this idea, we suggest the following alphabet reduction algorithm.

Given an alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ and a reduction threshold γ, we reduce $\mathcal{A}$ in the following manner:

$\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$
$p = n$
while $p > 1$
  find the combination C^* of p elements from $\mathcal{A}$ with minimum entropy $H(C^*)$
  if $H(C^*) < \gamma$
    merge the elements of C^*
    update $\mathcal{A}$
  else
    $p = p - 1$
  end
end

This alphabet reduction algorithm serves many purposes. First, as mentioned earlier, this algorithm splits the motion alphabet (Σ) obtained from the earlier step into two alphabets (Ξ and $\mathcal{K}$) so that the recovered hybrid strings can be viewed as I/O strings of a hybrid automaton. Second, this process facilitates noise reduction, which is naturally occurring when dealing with empirical data. Additionally, the reduction in the size of the alphabet makes the construction of automata more tractable. Finally, this process also leads to a stochastic interpretation of the feedback and interrupt functions, which is sometimes more natural than a deterministic interpretation. However, it should be noted that the computational burden associated with this algorithm may be quite high in that every mode combination must be computed.

From MDL Strings to Finite Automata

After applying the alphabet reduction algorithm presented earlier, our recovered string is of the form $k_{i_1}\xi_{j_1}k_{i_2}\xi_{j_2}\cdots$, where we can think of k_{i_q} as the output from the underlying finite automaton in state s_q, and ξ_{j_q} as the corresponding event that triggers a transition from state s_q to s_{q+1}. The question then becomes, can we recover this underlying automaton? And, is it unique? The answer to the second question is "no," and we will focus our attention on trying to recover minimal automata, but first we need to establish some notation.

An output automaton is a sextuple $\langle S, \Sigma, Y, s_0, T, h\rangle$, where S is the finite set of states, Σ is the input alphabet, Y is the output alphabet, $s_0 \in S$ is the initial state, $T \subseteq S \times \Sigma \times S$ is the set of allowable transitions, and $h : S \to Y$ is the output function. For our purposes, the input and output alphabet will be the finite set of interrupts (Ξ) and feedback laws ($\mathcal{K}$),

respectively. We define a path π as a finite alternating sequence, $s_{i_1}\sigma_{j_1}s_{i_2}\sigma_{j_2}s_{i_2}\cdots\sigma_{j_{n-1}}s_{i_n}$, of states and inputs, starting and ending with a state. We say that a path is executable on A if $s_{i_1} = s_0$, and each input transitions the state preceding it to the one following it, i.e., $(s_{i_q}, \sigma_{j_q}, s_{i_{q+1}}) \in T$ for all q. An I/O path π_y is an alternating sequence, $y_{\ell_1}\sigma_{j_1}y_{\ell_2}\sigma_{j_2}\cdots\sigma_{j_{n-1}}y_{\ell_n}$, of outputs and inputs, starting and ending with an output. We say that an I/O path is executable on A if there exists an executable path $s_{i_1}\sigma_{j_1}s_{i_2}\sigma_{j_2}\cdots s_{j_n}$ such that for all q, $h(s_{i_q}) = y_{\ell_q}$. Now, given an I/O path $\pi_y = y_{\ell_1}\sigma_{j_1}\cdots y_{\ell_n}$, the problem under consideration here is to find the smallest deterministic output automaton $A = \langle S, \Sigma, Y, s_0, T, h\rangle$ on which π_y is executable.

Note that for a given I/O path $\pi_y = y_{\ell_1}\sigma_{j_1}\cdots y_{\ell_n}$, there always exists at least one output automaton A on which π_y is executable. It is the automaton that jumps to a new state at each transition. This sequential output automaton has exactly n states. The set of automata that can execute π_y is thus nonempty, and there always exists a solution to the problem defined earlier.

In fact, this problem is related to the problem of producing minimal equivalent automata since one could consider applying a state reduction algorithm to the sequential output automaton derived earlier. However, this automaton is not necessarily complete, which is a necessary condition for applying such algorithms [4]. There is, however, an abundance of literature pertaining to the reduction of incompletely specified automaton. This problem is known to be NP-complete [18]. The various approaches for solving this problem can be categorized as either exact or heuristic based. The standard approach for this problem is based on enumeration of the set of compatible states and the solution of a binate covering problem [15]. A different approach for exact minimization not based on enumeration is presented in [16], and [19] presents heuristic-based algorithms that significantly reduce run time while obtaining correct results in most cases. Finally, what we are aiming for can also be cast in terms of finding the smallest automaton that simulates a particular sequential output automaton by using the terminology in [9].

As mentioned earlier, MODEbox allows the user to select between an optimal and suboptimal algorithm.

The optimal algorithm is an exhaustive search algorithm. The set of all consistent automata is progressively constructed by reading the I/O path from left to right. At the end of this search, the automaton with the fewest number of states is the optimal solution. As the length of the I/O path increases, the number of possible candidates quickly increases as well. Indeed, it is easy to show a superexponential relation for a worst-case scenario. To contain this explosion, we apply two heuristic modifications to the algorithm.

- We limit the memory resources so that only a fixed number of automata M can be stored. When this number has been reached, new candidate automata are automatically discarded.
- We set a maximum size $c_{\max}$. If an automaton has more than $c_{\max}$ states, it is discarded.

In MODEbox, we encode these heuristics in a high-level iterative algorithm that slowly increases the bounds until a solution is reached. Although this modification reduces the computation time, the problem remains NP-complete and quickly becomes numerically intractable for long I/O paths. For this reason, the user can specify a time limit after which, if no solution has been found, the algorithm returns a lower bound on the size of the optimal solution.

MODEbox allows the user to select between an optimal and suboptimal algorithm.

Instead of keeping up with all consistent automata (as done in the optimal algorithm), the suboptimal algorithm constructs a single automaton consistent with the given I/O path by greedily selecting one of them randomly. As before, the automaton is constructed by progressively reading the I/O path from left to right. At each iteration, we identified potential (consistent) candidates for the next state and chose one of them randomly. A new state is added only when previously created states cannot be used. This results in a small consistent automaton, but the random process in this selection cannot guarantee that the solution is optimal. However, this algorithm is significantly faster than the previous one. In fact, it has cubic complexity with respect to the length of the I/O path.

Robots and Ants

We now illustrate how MODEbox can be put to use in two particular examples, involving mobile robots and ants. In the first example, the system should recover a multimodal control strategy, given I/O data obtained when controlling a mobile robot using two distinct dynamic behaviors, whereas the other example is given by the observation of a biological system.

Control of Mobile Robots

For this example, originally reported in [1], we use a unicycle-type robot, i.e., its kinematics are given

$$\begin{aligned}\dot{x} &= v\cos\phi\\ \dot{y} &= v\sin\phi\\ \dot{\phi} &= \omega,\end{aligned}$$

where (x, y) is the position and ϕ is the heading of the robot. The translational and angular velocities (v, ω) are the control variables, and we quantize them according to $u \in \{(v,\omega)|\ v \in V, \omega \in \Omega\}$, where

$$\begin{aligned}V &= \{\text{slow, medium, fast}\},\\ \Omega &= \{\text{fast left, slow left, straight, slow right, fast right}\}.\end{aligned}$$

In a similar manner, the measurements made by the robot are sampled and quantized to produce an output string. We let

The robot is driven manually, and the resulting input-output string serves as input to MODEbox so that we can control the robot through the recovered control strategies.

$y \in \{(y^1, y^2, y^3) | y^1 \in Y^1, y^2 \in Y^2, y^3 \in Y^3\}$, where y^1 gives the distance to the closest obstacle, y^2 gives the relative angle to the closest obstacle, and y^3 gives the relative angle to the goal. By letting the angular quantization be given by the positions of the sensors on the sonar ring, as shown in Figure 8(a), we get

$$Y^1 = \{\text{close, medium, far}\},$$

$$Y^2 = \{1, 2, \ldots, 8\}, \quad Y^3 = \{1, 2, \ldots, 8\}.$$

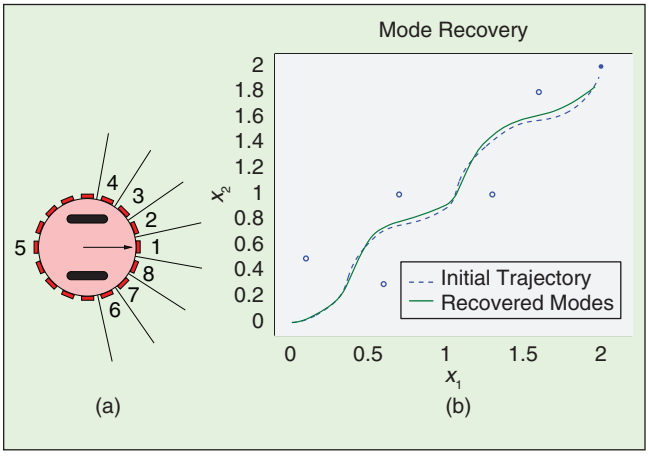

Figure 8. *(a) Angular quantization. (b) Effect of controlling the robot using a recovered mode string.*

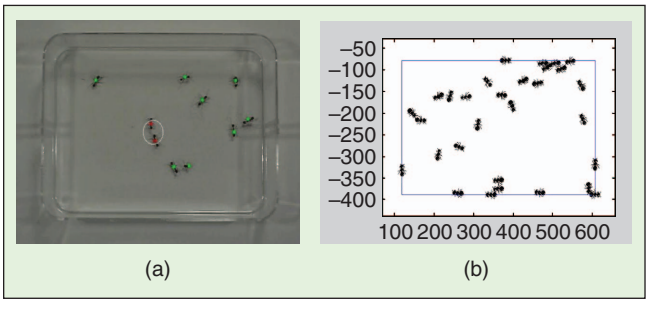

Figure 9. *(a) Ten ants are moving around in a tank. The circle around two ants means that they are docking or exchanging information. (b) Simulation environment depicting 30 simulated ants.*

In the experiment, we let the actual robot be controlled using

$$v = v_0 \min\{1, (d_{ob}/D)^2\}$$
$$\omega = C_{ob}(d_{ob})(\phi_{ob} + \pi - \phi) + C_g(\phi_g - \phi),$$

where D is a specified safety distance, d_{ob}, ϕ_{ob} is the distance and direction to the closest obstacle, ϕ_g is direction to the goal, and the gain $C_{ob}(d_{ob}) = 0$ if $d_{ob} \geq D$ and $(d_{ob} - D)/d_{ob}^3$ otherwise.

The robot is driven manually, and the resulting I/O string serves as input to MODEbox so that we can control the robot through the recovered control strategies. The results are shown in Figure 8(b), where the real execution (dashed line) is shown together with the effect of controlling the robot using the MODEbox tool.

Ants in a Tank

We now consider an example from [6], where ten ants (*Aphaenogaster cockerelli*) are placed in a tank with a camera mounted on top [Figure 9(a)]. A 52-s movie is used to extract the Cartesian coordinates x and y and the orientation θ of every ant every 33 ms using a vision-based tracking software. This experimental setup is provided by Tucker Balch and Frank Dellaert at the Georgia Institute of Technology BORG Lab [13], [21].

From this experimental data, an I/O string at each sample time k is found for each ant i as follows:

- the input u_k is given by (u_k^1, u_k^2), where u_k^1 is the quantized angular velocity, and u_k^2 is the quantized translational velocity of the ant i at time k
- the output y_k is given by (y_k^1, y_k^2, y_k^3), where y_k^1 is the quantized angle to the closest obstacle, y_k^2 is the quantized distance to the closest obstacle, and y_k^3 is the quantized angle to the closest goal of ant i at time k.

An obstacle is either a point on the tank wall or an already visited ant within the visual scope of ant i, and a *goal* is an ant that has not been visited recently.

These data are fed to MODEbox, and the resulting control programs can be used to simulate ant behavior. Figure 9(b) shows an example of this when 30 ants are simulated based on the recovered hybrid control strategy.

Conclusions

In this article, we introduce the MODEbox tool for automatically producing hybrid, multimodal control programs from data. In particular, given an I/O string, four distinct operational units are introduced.

- *Preprocessing*: The real-valued I/O strings are transformed into strings of symbols through quantization, Lebesgue sampling, and encoding operations.
- *MDL capturing*: Low-complexity strings of symbolic feedback-interrupt pairs are produced that are consistent with the empirical data.
- *MDL to automata*: Small finite automata are produced in such a way that outputs correspond to feedback mappings, and transition events correspond to interrupts.
- *Simulation or execution*: Simulated or real systems can be controlled using the obtained hybrid control strategy.

The application of the MODEbox tool is illustrated on two different examples involving robots and ants.

Acknowledgments

We thank Tucker Balch and Frank Dellaert for the ant data that were made available through the BORG Lab at the Georgia Institute of Technology. Part of the work on which this article is based was conducted by Adam Austin and Johan Isaksson, and their participation is greatly appreciated. This work was supported by the National Science Foundation through the programs EHS NSF-01-161 (Grant 0207411) and ECS NSF-CAREER award (Grant 0237971).

Keywords

Multimodal control, hybrid systems, mobile robots.

References

[1] A. Austin and M. Egerstedt, Mode reconstruction for source coding and multimodal controlin *Hybrid Systems: Computation and Control*, O. Maler and A. Pnueli, Eds. Prague, The Czech Republic: Springer-Verlag, 2003, pp. 36–49.

[2] K. J. Åström and B. M. Bernhardsson, "Comparison of Riemann and Lebesgue sampling for first order stochastic systems," in *Proc. IEEE Conf. Decision and Control*, Las Vegas, NV, 2002, pp. 2011–2016.

[3] T. Balch and R. C. Arkin, "Behavior-based formation control for multi-robot teams," *IEEE Trans. Robot. Automat.*, vol. 14, pp. 926–939, Dec. 1998.

[4] R. G. Bennetts, J. L. Washington, and D. W. Lewin, "A computer algorithm for state table reduction," *Radio Electron. Eng.*, vol. 42, no. 4, pp. 513–520, 1972.

[5] R. W. Brockett, "On the computer control of movement," in *Proc. IEEE Conf. Robotics and Automation*, New York, 1988, pp. 534–540.

[6] F. Delmotte and M. Egerstedt, "Reconstruction of low-complexity control programs from data," in *Proc. IEEE Conf. Decision and Control*, Atlantis, Bahamas, 2004, pp. 1460–1465.

[7] F. Delmotte, M. Egerstedt, and A. Austin, "Data-driven generation of low-complexity control programs," *Int. J. Hybrid Syst.*, vol. 4, no. 1-2, pp. 53–72, 2004.

[8] M. Egerstedt and R. W. Brockett, "Feedback can reduce the specification complexity of motor programs," *IEEE Trans. Automat. Contr.*, vol. 48, pp. 213–223, Feb. 2003.

[9] A. Girard and G. J. Pappas, "Approximate bisimulations for nonlinear dynamical systems," in *Proc. 44th IEEE Conf. on Decision and Control*, 2005, pp. 684–689.

[10] D. Grünbaum, S. Viscido, and J. K. Parrish, "Extracting interactive control algorithms from group dynamics of schooling fish," in *Proc. Block Island Workshop on Cooperative Control*, V. Kumar, N. E. Leonard, and A. S. Morse, Eds. New York: Springer-Verlag, 2004.

[11] J. E. Hopcroft and G. Wilfong, "Motion of objects in contact," *Int. J. Robot. Res.*, vol. 4, no. 4, pp. 32–46, 1986.

[12] A. Lj. Juloski, W. P. M. H. Heemels, G. Ferrari-Trecate, R. Vidal, S. Paoletti, and J. H. G. Niessen, "Comparison of four procedures for the identification of hybrid systems," in *Proc. 8th Int. Workshop, Hybrid Systems: Computation and Control*, Mar. 2005, pp. 354–369.

[13] Z. Khan, T. Balch, and F. Dellaert, "An MCMC-based particle filter for tracking multiple interacting targets," Georgia Inst. of Technology, Atlanta, GA. Tech. Rep. GIT-GVU-03-35, Oct. 2003.

[14] V. Manikonda, P. S. Krishnaprasad, and J. Hendler, "Languages, behaviors, hybrid architectures and motion control," in *Mathematical Control Theory*, J. Willems and J. Baillieul, Eds. New York: Springer-Verlag, 1998, pp. 199–226.

[15] M. Paull and S. Unger, "Minimizing the number of states in incompletely specified state machines," *IRE Trans. Electron. Comput.*, vol. EC-8, pp. 356–367, Sept. 1959.

[16] J. M. Pena and A. L. Oliveira, "A new algorithm for exact reduction of incompletely specified finite state machines," in *Proc. Int. Conf. Computer Aided Design*, 1998, pp. 482–489.

[17] V. Pavlovic, J. M. Rehg, and J. MacCormick, "Learning switching linear models of human motion," in *Advances in Neural Information Processing Systems 13* (NIPS*2000), pp. 981–987.

[18] C. F. Pfleeger, "State reduction in incompletely specified finite state machines," *IEEE Trans. Comput.*, vol. C-22, pp. 1099–1102, 1973.

[19] J. K. Rho, G. Hachtel, F. Somenzi, and R. Jacoby, "Exact and heuristic algorithms for the minimization of incompletely specified finite state machines," *IEEE Trans. Computer-Aided Design*, vol. 13, no. 2, pp. 167–177, 1994.

[20] *GRITS lab* (2007). [Online]. Available: http://gritslab.ece.gatech.edu/MODEbox.html

[21] *BORG lab* (2006). [Online]. Available: http://borg.cc.gatech.edu

Florent Delmotte received a Diplôme d'Ingénieur from the Ecole Supérieure d'Electricité (Supélec), Gif-sur-Yvette, France, in 2003. Since then, he has been a graduate research assistant in the Department of Electrical and Computer Engineering at the Georgia Institute of Technology, Atlanta, Georgia, where he received an M.S. degree in 2003. He is now a Ph.D. candidate. His research interests include hybrid systems, multimodal estimation, linguistic control of mobile robots, and optimal control.

Tejas R. Mehta is pursuing his Ph.D. degree at the Georgia Institute of Technology, where he received his M.S. degree in 2004 and his B.S. degree in 2002 in electrical and computer engineering. He is currently working as a graduate research assistant at the Georgia Robotics and Intelligent Systems Laboratory. His research interests include optimal control of hybrid systems, multimodal control, and linguistic control of mobile robots.

Magnus Egerstedt is an associate professor in the School of Electrical and Computer Engineering at the Georgia Institute of Technology, where he has been on the faculty since 2001. He received the M.S. degree in engineering physics and the Ph.D. degree in applied mathematics from the Royal Institute of Technology, Stockholm, in 1996 and 2000, respectively. He also received a B.A. degree in philosophy from Stockholm University in 1996. From 2000 to 2001, he was a postdoctoral fellow at the Division of Engineering and Applied Science at Harvard University. His research interests include optimal control and modeling and analysis of hybrid and discrete event systems, with an emphasis on motion planning and control of (teams of) mobile robots.

Address for Correspondence: Magnus Egerstedt, School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA. Phone: +1 404 894 3484. E-mail: magnus@ece.gatech.edu.