MDLn: A Motion Description Language for Networked Systems

Patrick Martin, Jean-Pierre de la Croix, and Magnus Egerstedt
pmartin@ece.gatech.edu, jdelacroix@gatech.edu, magnus@ece.gatech.edu
Georgia Institute of Technology, Atlanta, GA 30332, USA

***Abstract*— In this paper we extend the definition of a Motion Description Language (MDL) to networked systems. This new construction (MDLn) supports inter-agent specification rules as well as desired network topologies, enabling us to specify high-level control programs for group interactions. In particular, MDLn-strings specify multi-modal executions of the system through a concatenation of modes. Each mode in the MDLn-string is a triple, specifying a control law, interrupt conditions, and desired network dependencies. In addition to proposing MDLn as a specification language for networked systems, we also give an architecture in which MDLn strings can be effectively parsed and executed in multi-robot applications.**

I. Introduction

Motion Description Languages (MDL) [1], [2], [3] are formal languages in which control programs can be specified for multi-modal systems. Such programs are useful for encoding the decomposition of complex control tasks into building-blocks, concatenated together to achieve complex control objectives, encountered, for example in robotics [4], [5], [2], manufacturing [6], and sensor networks [7].

In this paper we extend the definition of an MDL to make it applicable to networked systems in which not only the control laws, but also the desired network topologies, are to be specified and changed dynamically. In particular, we focus on multi-robot systems, in which a collection of mobile agents are to achieve some coordinated goal.

Previous work in this area of inquiry has mainly been conducted by Klavins and co-workers, first through the Communication and Control Language (CCL) [8] and later through Embedded Graph Grammars [9]. CCL is a high-level language in which asynchronous, interacting systems can be modeled and programmed. What is appealing about CCL is that coordinated control tasks can be programmed in a manner akin to standard programming languages. However, it does not provide the structure sought in this paper that explicitly addresses just what the essential components should be when solving coordinated, multi-agent control problems.

EGGs, on the other hand, do address this issue, and they are easy to use when the network consist of large collections of identical (or nearly identical) agents. In fact, EGGs have mainly been applied when the desired, combinatorial interaction topologies are highly complicated but the agent dynamics are straightforward, as is the case with self assembly systems [9].

In contrast to this, we focus on systems in which the networks are heterogeneous (the different agents may take on different roles) and where the interaction topologies may very well be specified *a priori*. An example scenario would be leader-based formation control.

The outline of this paper is as follows: In Section II, we recall the basic operation of Motion Description Languages, followed by their extensions to networks (MDLn), in Section III. Section IV focuses on the system architecture needed to support MDLn, while a number of example application scenarios are given in Section V. Section VI contains the conclusions.

II. Motion Description Languages

MDLs, first defined in [1], are strings of control modes that define a hybrid control system. Each mode applies an open or closed loop control law, u, for a given duration to a system modeled by

$$\begin{aligned} \dot{x} &= f(x,u),\ x \in \mathcal{X},\ u \in \mathcal{U} \\ y &= h(x), y \in \mathcal{Y}, \end{aligned} \tag{1}$$

until some switching condition is satisfied. The original formulation in [1] focused on the problem of controlling a manipulator arm in an unstructured environment; however, this approach to controlling hybrid systems lends itself well to other robotics applications.

In [2], an extended Motion Description Language, MDLe, was defined to support sensor driven interrupt functions. These interrupt functions, defined by the mapping $\xi_i : \mathcal{Y} \rightarrow \{0,1\}$, take sensor output from the mobile robot to determine mode switch conditions. This modification results in the modes, or *atoms*, taking the form (κ_i, ξ_i, T_i), where the index, i, indicates which mode in the string is running and κ_i represents the control law produced by the mapping $\kappa_i : \mathcal{Y} \rightarrow U$. This control is applied to the model (1) until ξ_i transitions to 1 *OR* the timer, T_i, fires. In [3] the interrupt and timer were composed into the same function via a logical *OR*, resulting in atoms that were given by pairs, (κ_i, ξ_i).

In this paper we are interested in controlling multiple robots and so we have to augment the current MDL framework to also encompass agent interaction specifications in the network. It should be mentioned that there has been recent work on modifying Motion Description Languages to allow for *group* atoms [10], which are special atoms that allow for a global control and interrupt function. These modified MDLs have been successfully applied to formation control. Additionally, in [11] MDLe strings are composed into system behaviors, created by the individual mode sequences of the agents involved. In this paper we take an alternative approach by formulating a new mode structure in order

to encode the communication relationships necessary for agent collaboration. Before we can specify this new mode structure, some preliminary definitions of the system model and network topology must be presented.

III. MDL for Multi-agent Systems

In order to extend MDLs for their use in networked systems we let each agent's dynamics be given by,

$$\begin{aligned} \dot{x}_i &= f_i(x_i, u_i) \\ y_i &= h_i(x_i) \\ s_i &= g_i(x_i, y_i) \end{aligned} \tag{2}$$

where $x_i \in \mathcal{X}_i \subseteq \mathbb{R}^n$, $y_i \in \mathcal{Y}_i \subseteq \mathbb{R}^p$, and $s_i \in \mathcal{S}_i \subseteq \mathbb{R}^m$ $(m \leqslant p+n)$. The way these entities should be understood is as follows: the current state of agent-i, x_i, determines the local information produced by its sensors, y_i. Additionally, agent-i transmits its *shareable information*, s_i, by mapping its state and sensor output into a vector via the function $g_i : \mathcal{X}_i \times \mathcal{Y}_i \to \mathcal{S}_i$. Note that although this product of state and output spaces may not be needed, the inclusion of $\mathcal{Y}_i$ makes the environmental dependence of shared information more explicit. This information may then be transmitted through the network to a desired neighbor.

For example, say agent-i, which we denote as a_i, is a mobile robot with state $x_i = [x_{i,1}\ x_{i,2}\ x_{i,3}]^T$, where $(x_{i,1}, x_{i,2})$ is the Cartesian coordinate of the robot and $x_{i,3}$ its orientation. Additionally, let a_i have a four sensor sonar-array, where each sonar produces two data points for each reading, i.e. $p_{i,j} \in \mathbb{R}^2$. Then the output vector of a_i is $y_i = [p_{i,1}\ p_{i,2}\ p_{i,3}\ p_{i,4}]^T \in \mathbb{R}^8$. If a_i plans to share its heading, $x_{i,3}$, and the forward sensor outputs, $p_{i,1}$ and $p_{i,2}$, then the following shareable information vector is produced by the mapping s_i:

$$s_i = \begin{bmatrix} x_{i,3} \\ p_{i,1} \\ p_{i,2} \end{bmatrix}.$$

This function facilitates the sharing of only the information that a_i wishes to reveal to members of its network. However, agents do not share arbitrarily, since passing the data to anyone in a network would cause unnecessary traffic.

A. Agent Buddies

What is missing from the MDL formulation when it comes to networked systems is the notion of agent-to-agent interactions. In particular, we need to be able to specify what neighboring agents (within communication range) the individual agents should interact with. We formalize this concept in MDLn by letting agents define their *preferred* neighbors, or *buddies*. Agents in a network select their desired neighbors (that may or may not be available in the network) as "static" buddies, denoted $\beta_s^i \subseteq 2^{\mathcal{N}}$, where $\mathcal{N} = \{1, \cdots, N\}$ and $N \in \mathbb{N}$ is the total number of agents in the network. Additionally, the specification may call for "dynamic" buddies (denoted β_d^i) to be added to this buddy list.

We require a clear formulation of the agent network in order to properly define the notion of buddies. We define the *egocentric network* for agent-i as any set of agents, $\mathcal{W}_i$, which we encode with the mapping $w_i : \mathcal{Y}_i \to 2^{\mathcal{N}}$. Therefore, agent-$i$'s network is determined by examining its sensor data to measure if any agents are within physical communication range. Robotic platforms may use their network devices, where low level signaling automatically determines communication range, or some combination of sensors, like RFID or vision, to determine their network members.

Then the dynamic buddies are a subset of all members of the network, β_d^i, resulting from the mapping $b^i : \mathcal{W}_i \times \mathcal{Y}_i \to 2^{\mathcal{W}_i}$. This definition of β_d^i states that the set of agent-i's dynamic buddies is a function of the members of agent-i's network and agent-i's sensor readings of these members. Consequently, the total set of agent-i's available buddies is dependent on the current static and dynamic buddies:

$$\beta^i = (\beta_s^i \cap \mathcal{W}_i) \cup \beta_d^i \subseteq 2^{\mathcal{N}}. \tag{3}$$

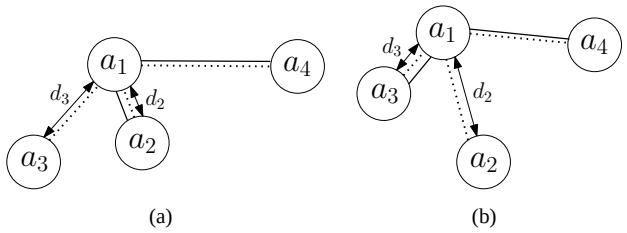


Fig. 1. An example of the network relationships for robot a_1. 1(a) shows that a_2 is the closest neighbor; however, in 1(b) a_3 has passed a_2 and is now a_1's new dynamic buddy.

This encoding of a_i's buddies is made more concrete by examining a small network of robots. Fig. 1 shows an example of a particular network view centered around robot a_1. All three of a_1's neighbors are in communication range, illustrated by the dotted lines. We choose arbitrarily that a_4 should be a static buddy, i.e. $\beta_s^1 = \{a_4\}$. Additionally, we create a dynamic buddy relationship such that a_1 also prefers the closest agent within communication range. This choice for the dynamic buddy is also arbitrary, since we could easily define some other metric to decide which agent could be a dynamic buddy.

Fig. 1(a) shows the initial positions of the four agents. In this case a_1 measures the distance between itself and a_2 as d_2 and the distance to a_3 as d_3. Since $d_2 < d_3$, a_2 is chosen as the current dynamic buddy: $\beta_d^1 = \{a_2\}$. Applying the buddy relationship of (3), the buddy list of a_1 is

$$\beta^1 = (\{a_4\} \cap \{a_2, a_3, a_4\}) \cup \{a_2\} = \{a_2, a_4\},$$

and is visualized by the solid lines between a_1 and a_2, a_4. Note that if a_2 wanders further away and a_3 approaches a_1 (Fig. 1(b)), the measured distances change and consequently a_3 becomes the new dynamic buddy.

B. Agent Roles

Although the buddy definition introduced in III-A properly describes who an agent prefers to communicate with, there

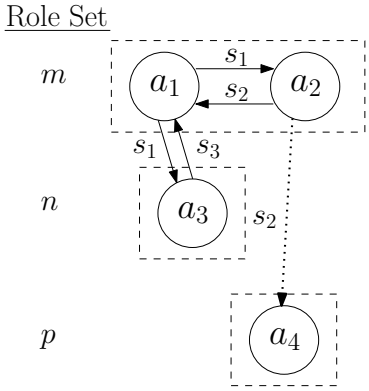


Fig. 2. An example of applying roles to the agents from section III-A.

is no specification of restrictions in a hierarchical network. Multi-agent systems may be composed of heterogeneous entities with various *roles*. Subsequently, these roles further specify communication relationships among agents in the network.

We define agent-i's role as a static value resulting from the mapping $r : \mathcal{N} \to \mathcal{R}$, where $\mathcal{R} \subset \mathbb{N}$ and is finite. This value determines the communication relationships among all other agents in the network via the following rules: for any agents a_i and a_j, $i \neq j$:

- R1: if $r(i) > r(j)$ then a_i may receive shared information from a_j.
- R2: if $r(i) = r(j)$ then a_i and a_j have no constraints on sharing information.
- R3: if $r(i) > r(j)$ *and* $a_j \in \beta^i$ then a_i and a_j have no constraints on sharing information.

We specify the role of each agent in advance and then the role comparison rules are applied at runtime. For example, using the setup described in III-A, we let $r(a_1) = r(a_2) = m$, $r(a_3) = n$, and $r(a_4) = p$ with $m > n > p$. A visualization of the hierarchy is shown in Fig. 2. Each arrow in the diagram shows the direction of information flow according to each agent's buddy list and their role set. Let the agents in this example have the following buddy list assignments:

$$\begin{aligned}\beta^1 &= \{a_2, a_3\}\\ \beta^2 &= \{a_1\}\\ \beta^3 &= \{a_1\}\\ \beta^4 &= \{a_2\}.\end{aligned}$$

The diagram shows that a_1 may pull information from a_2 since they share the same role class and from a_3 since it "outranks" the agent. Additionally, a_2 may get shareable information from a_1 and a_3 is allowed access to a_1's information since it resides in a_1's buddy list. The only agent that is left out is a_4. This agent is in the lowest role class and is not allowed to get a_2's shareable information (shown by the dotted line); however, if $a_4 \in \beta^2$ access to a_2's shared information would be granted.

C. MDLn Specification

As established by the model (2), each agent shares its data based on the value of $s_i = g_i(x_i, y_i)$. Consequently, if agent-i has k buddies, then the total shared information of agent-i's buddies is defined as

$$\hat{\mathcal{S}}_i = \mathcal{S}_{\beta^i(1)} \times \cdots \times \mathcal{S}_{\beta^i(k)},$$

where $\beta^i(\cdot)$ indexes agent-i's buddy list. The object $\hat{\mathcal{S}}_i$ can be thought of as a vector of shared information, i.e. $\hat{\mathcal{S}}_i \in \mathbb{R}^{km}$, held locally at agent-$i$. Agent-$i$ can now use the shared information of these agents when making control and interrupt decisions.

Using all of the above definitions, the control and interrupt functions may be modified as follows. The control depends on the state and sensor feedback of agent-i in addition to the information from all buddies of agent-i,

$$\kappa^i : \mathcal{X}_i \times \mathcal{Y}_i \times \hat{\mathcal{S}}_i \times \mathbb{R}^+ \to \mathbb{R}^m.$$

Additionally, the interrupt function uses the same local and shared information as

$$\xi^i : \mathcal{X}_i \times \mathcal{Y}_i \times \hat{\mathcal{S}}_i \times \mathbb{R}^+ \to \{0, 1\}.$$

We thus define a MDLn language as a set of strings (concatenations) made up from triples, (κ, ξ, β), where κ is a control law, ξ is an interrupt function, and β is a buddy list.

D. Parser

In this section we discuss a centralized MDLn parser that uses a grammar [12] to generate the valid MDLn strings from a script file. In addition to generating control modes based on the definition in section III-C, the parser must assign roles and buddies, as well as check them for relationship consistency. For example, in the same way that traditional programming language compilers check for variable declarations, the MDLn parser ensures that any buddy used by a control mode exists. Also, it verifies that buddies referenced in modes satisfy the role requirement for that particular agent.

Generating our MDLn programs requires a grammar so that roles and modes may be parsed to allow for consistency checks and MDLn string distribution. We define the grammar for MDLn programs as

$$G = (\{P, R, S, I, M\}, \{r, k, x, b\}, \hat{P}, P)$$

with the following productions $\hat{P}$,

$$\begin{aligned}P &\to R^\star\ S^+\\ R &\to I\ r\\ S &\to I\ M^+\\ M &\to k\ x\ b.\end{aligned}$$

The nonterminal P is the start symbol for an MDLn program and is produced by the nonterminals R and S which stand for the roles and MDLn strings, respectively. Therefore, an MDLn program must have a list of roles followed by a list of strings.

Note that this formulation does not *require* roles in every MDLn program since the symbol R uses the $(\star)$ operator; however the $(+)$ operator does require at least one MDLn string to be a valid program. The roles are produced by the nonterminal representing an identifier, I, which is similar to a variable name in standard programming languages, followed by the role map terminal, r.

Finally, the MDLn string productions consist of an identifier, indicating which agent is using the MDLn string, and a list of at least one MDLn mode, M. This nonterminal M is made by concatenating the terminals $k, x,$ and b which stand for the triple (κ, ξ, β) seen in section III-C.

These productions specify the syntax of how a valid MDLn script file, or program, should be structured. The parser can then use these rules to run through a given program validating necessary references (i.e. controls, interrupts, and buddies) and determining role inconsistencies. These static checks enforce the rules proposed in III-B at compile time, and the parser can reject the MDLn program, remove any illegal role usages, or attempt to correct the error. For example, say the parser is given a program:

```
agent1 2
agent2 0

agent1 (k1 x1 {agent2})
agent2 (k2 x1 {agent1})
```

We see that the first two lines make up the production rule R, where the nonterminals, I, are `agent1` and `agent2` and the role assignments of the robots, r, are 2 and 0, respectively. The bottom two lines make up two S productions, where each one has the identifier I and one mode nonterminal, M. These mode nonterminals are made up of the three MDLn terminal symbols, $k, x,$ and b. Note that this example has the additional symbols `(`, `)`, {, and }, which are used to make the script easier to read.

The parser stores the identifier of the first S production, `agent1`, checks the availability of the `k1` and `x1` functions for that particular agent and finally stores the reference to `agent2`. The next production creates a mode string for `agent2`, which uses a different control function, `k2`, and also references `agent1`. When the parser reaches the end of this program, it then checks the buddy consistency, which in this case is valid since both agents have been identified and exist in the program. Additionally, the static role consistency check passes since `agent2` references `agent1`, *and* `agent2` is in `agent1`'s buddy list .

IV. System Architecture

In [2], a system architecture was prescribed for using MDLs on single robots. Our architecture incorporates this; however, we have designed additional components that facilitate the new features of MDLn. An illustration of the architecture in seen in Fig. 3.

The MDLn architecture of agent-i is made up of several primary components. At the highest level is the *MDLn Driver*, which manages the state of the agent and enables

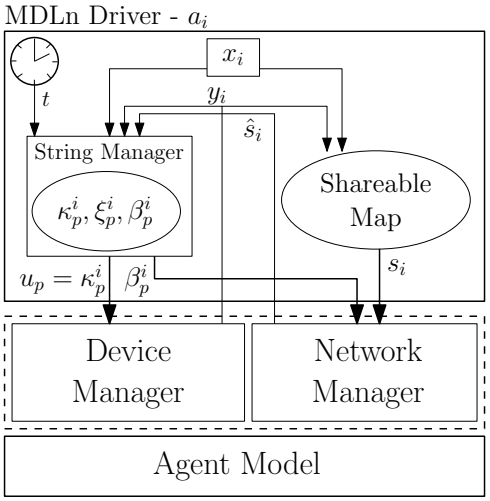


Fig. 3. The architecture for a particular MDLn enabled agent, a_i. This agent is currently executing the p^{th} mode, $(\kappa_p^i, \xi_p^i, \beta_p^i)$ of a_i's mode string.

the interpretation of MDLn strings. This component drives the agent by choosing the proper mode to run and creating the shareable information vector of the agent. The next layer down is called the hardware abstraction layer, or HAL. The HAL provides the connection between the high level control and low level implementation details. It manages sensors, actuators, and communications devices. Finally, the HAL communicates with the lowest level, the Agent Model, which contains any system information about devices, simulated or real.

Internally, the MDLn Driver has a String Manager, which handles the interpretation of "compiled" MDLn strings. It runs off of the system clock, which allows for the timer interrupt capabilities seen in previous MDL architectures. Additionally, it receives all necessary information for applying agent-i's current control mode, $(\kappa_p^i, \xi_p^i, \beta_p^i)$, where the index p is some arbitrary index into agent-i's mode string. The String Manager then outputs the current control signal and the current set of buddies in that particular control mode; additionally, it computes the interrupt function to determine if the next mode in the string should be executed.

The control signal is received by the Device Manager and the buddy list is received by the Network Manager. The MDLn Driver uses the Shareable Information module to generate information for agent-i's network buddies. The Device Manager takes the control input, κ_p^i, and calls the appropriate actuator methods of the agent model. At the same time, the Device Manager serves the String Manager the current sensor data, y_i.

The Network Manager in the HAL uses the buddy list, β_p^i, to enforce any communication requirements specified by the MDLn program. It also sends the shareable information of agent-i, s_i, as messages to all agents in agent-i's role set. Finally, the Network Manager must serve the shareable information vector of agent-i's current buddies, $\hat{s}_i$, to the String Manager so that control laws and interrupt functions may use the data in their execution.

Although this design choice for the architecture is clearly not unique, it is well suited for the goals of the MDLn framework. Our architecture is designed so that the low level may be either a robotic platform or a simulation model. Consequently, the architecture allows for mixed networks of MDLn enabled hardware and software agents.

V. Application of MDLn

To show that multi-agent behaviors may be modeled and implemented within the MDLn framework, we present two examples of agents executing MDLn programs. The first, consensus, is a standard multi-agent control algorithm for collecting a set of agents at their centroid. The second example is a more complicated program which takes full advantage of the new features of MDLn.

A. MDLn Consensus

In the consensus problem, each agent has access to relative information about its neighbors within some distance Δ. In other words, the network is a time varying set $N(t)$. In the standard formulation the agents have dynamics

$$\dot{x}_i = -\sum_{j \in N(t)} (x_i - x_j),$$

which result in the agents converging to the centroid of their positions as long as the network stays connected. We can take these dynamics and encapsulate them within an MDLn mode via the control function

$$\kappa_c^i = -\sum_{j \in \beta^i} (x_i - x_j). \tag{4}$$

To make matters more precise, let there be three agents (a_1, a_2, a_3) with individual control actions taking the form of the dynamics in (4). Let each robot be equipped with sensors that can detect distances to obstacles. The information generated by these sensors can be used to define an interrupt function, ξ_{obs},

$$\xi_{obs} = \begin{cases} 1, & h_i(x_i) < D \\ 0, & otherwise \end{cases},$$

where D is some constant threshold value.

Letting each agent have one single consensus mode results in the sample MDLn program:

$$a_1 : (\kappa_c, \xi_{obs}, \{a_2, a_3\})$$
$$a_2 : (\kappa_c, \xi_{obs}, \{a_1, a_3\})$$
$$a_3 : (\kappa_c, \xi_{obs}, \{a_1, a_2\})$$

where the third term of each triple denotes the set of static buddies of that particular agent. This program generates a single-mode MDLn string for each of the three agents, where each agent performs consensus until it detects an obstacle. Consequently, when the obstacle detection interrupt, ξ_{obs}, fires, an agent will cease operation since there are no more modes in the MDLn string to execute.

In this example we let the roles of all agents be equal, i.e. $r(a_1) = r(a_2) = r(a_3)$. Therefore, the MDLn parser would accept this program since its syntax structure is valid and the usage of the agent references are consistent with the role sets. Moreover, at runtime the program is dynamically consistent since all agents are in the same role class, satisfying R1 in section III-B.

B. A Complex Program

The MDLn formulation of consensus showed a simple example of the usage of MDLn. In contrast to that example, we now consider an example which uses all of the features of MDLn to prescribe a more complex behavior of a multi-agent system.

Again, let there be three robots (a_1, a_2, a_3) with the following role assignments:

$$r(a_1) = 2$$
$$r(a_2) = r(a_3) = 1.$$

Each robot has their own set of motion primitives, made up of the following functions, $\mathcal{K}^i = \{\kappa_f, \kappa_a, \kappa_{gtg}\}$. The function κ_f defines a controller that follows a moving point in the Cartesian plane at some constant following distance. Additionally, the function κ_a defines a control primitive that avoids an obstacle, which can be implemented with a basic potential field algorithm. A robot can use the controller, κ_{gtg}, to move towards a static goal, also in the Cartesian plane.

Additionally, the robots have a set of interrupt functions, $\Xi^i = \{\xi_{obs}, \xi_{clr}\}$ which are the obstacle detected interrupt defined in section V-A and a new interrupt, ξ_{clr},

$$\xi_{clr} = \begin{cases} 1, & h_i(x_i) \geqslant D \\ 0, & otherwise \end{cases},$$

respectively. Using these control and interrupt functions, we create the following MDLn program:

$$a_1 : (\kappa_{gtg}, \xi_{obs}, \{a_3\})(\kappa_a, \xi_{clr}, \{a_3\})$$
$$a_2 : (\kappa_f, \xi_{obs}, \{a_1\})(\kappa_a, \xi_{clr}, \{a_1\})$$
$$a_3 : (\kappa_f, \xi_{obs}, \{a_1, \chi\})(\kappa_a, \xi_{clr}, \{a_1\}),$$

where we use the symbol χ for representing the "closest neighbor."

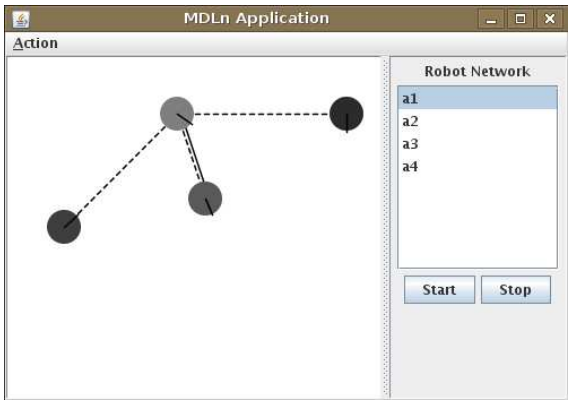
This program can be interpreted in the following way. We see that a_1 has a "leadership" role since its role value is larger than that of a_2 and a_3. This agent will start off moving towards the goal point until an obstacle is detected by itself *or* a_3, which is shown by the buddy dependence in the first mode, $(\kappa_{gtg}, \xi_{obs}, \{a_3\})$. Then, a_1 will switch into an obstacle avoidance behavior, and will stop when itself or a_3 is clear of obstacles. Note that the buddy dependence on a_3 in this mode operates on the assumption that the network will support this action in its implementation.

Additionally, for both static and dynamic buddy dependence, the controllers and interrupts must be well defined when the shareable information vector is missing certain buddy information. In this case, the ξ_{obs} function should be able to execute *at least* on a_1's local information, y_1.

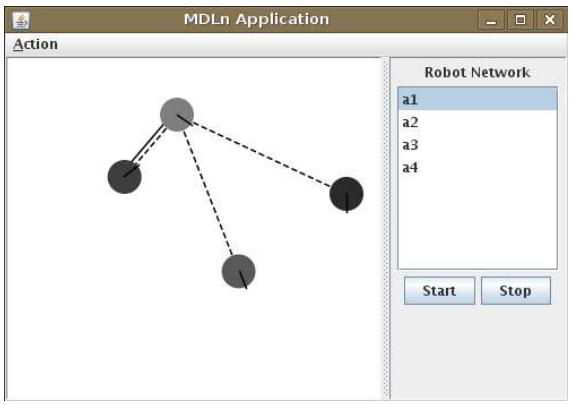
The second agent, a_2, starts off following a_1 and will do so until itself or a_1 detects an obstacle (similarly to a_1's first mode). It will also avoid the obstacle until it is clear or a_1 is clear. Finally, a_3's mode string makes the robot follow a_1 *or*

its closest buddy, which is determined from its set of dynamic buddies, β_d^3. This robot following mode will continue until an obstacle is detected locally or by either a_1 or χ. It will then avoid the obstacle until itself or a_1 is clear of obstacles.

This particular program brings up the importance of role consistency in MDLn. At parse time this program will have inconsistent role usage due to a_2 referencing a_1, which violates R1 in section III-B. Consequently, a_2 will not operate on its MDLn string; however, it may be possible to place a_2 in β^1, but that choice is left to the designer of the motion program. Note, also, that a_3 depends on a_1 *and* and its closest buddy in the first mode. This dependency works in this program since a_3 satisfies all role set rules, i.e. $r(a_2) = r(a_3)$ and $a_3 \in \beta^1$. The enforcement of this rule occurs within each agent's Network Manager, which is fed MDLn buddy dependencies when modes are executed, as described in section IV.



(a)



(b)

Fig. 4. A visualization of the software system that manages the low level architecture of MDLn.

C. Software Implementation

Our low level architecture has been implemented using Java and Player [13]. This software manages each robot's current network ($\mathcal{W}_i$) as well as dynamic buddies. Screenshots of the software are seen in Fig. 4. These images show a similar example to the one discussed in section III-A, where the buddies of a_1 change when a_3 moves closer within range than a_2. Fig. 4(a) shows the visualization of a_1's network (dotted lines) and dynamic buddies (solid lines). At the start, a_2 is the closest network member, and so a_1 lists a_2 as a buddy. However, in Fig. 4(b), a_3 has approached more closely to a_1 and a_2 has wandered too far away. Note that a_4 is not a buddy in this simulation since the MDLn Driver, which pushes static buddies to the Network Manager, has not been implemented yet.

VI. Conclusions

In this paper we extend Motion Description Languages to incorporate networked control aspects. In particular, we define MDLn as a concatenation of triples (κ, ξ, β), where the novel aspect is β, which encodes the *buddies* on which the control law operates. We show how to apply MDLn in a number of example scenarios, as well as discuss the architectural and simulation issues.

Acknowledgements

This work was supported in part by the U.S. Army Research Office Grant 99838 and in part under a contract with the National Aeronautics and Space Administration.

References

[1] R.W. Brockett. On the Computer Control of Movement. In *Proceedings of of the 1988 Conf. of Robotics and Automation*, pages 534-540, April 1988.

[2] V. Manikonda, P.S. Krishnaprasad, J. Hendler. Languages, behaviors, hybrid architectures and motion control. In J.C. Willems J. Baillieul, editor, *Mathematical Control Theory*, pages 199-226. Springer 1998.

[3] D. Hristu-Varsakelis, M. Egerstedt, and P.S. Krishnaprasad. On the structural complexity of the motion description language MDLe. In *42^{nd} IEEE Conference on Decision and Control*. 2003, pages 3360-3365.

[4] R. Arkin. *Behavior-Based Robotics*, MIT Press, 1998.

[5] M. Egerstedt. Behavior Based Robotics Using Hybrid Automata. *Lecture Notes in Computer Science: Hybrid Systems III: Computation and Control*, pp. 103-116, Pittsburgh, PA, Springer-Verlag, March 2000.

[6] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Norwell, MA, 1999.

[7] M. Zavlanos and G. Pappas. Distributed connectivity control of mobile networks. In *46^{th} IEEE Conference on Decision and Control*. 2007, pages 3591-3596.

[8] E. Klavins. A Language for Modeling and Programming Cooperative Control Systems, *Proceedings of the International Conference on Robotics and Automation*, May 2004, pp. 3403-3410.

[9] J.M. McNew and E. Klavins. Locally Interacting Hybrid Systems with Embedded Graph Grammars. In *45^{th} IEEE Conference on Decision and Control*. 2006, pages 6080-6087.

[10] F. Zhang, M. Goldgeier, P.S. Krishnaprasad. Control of Small Formations Using Shape Cooridinates. In *Proceedings of the International Conference on Robotics and Automation*, September 2003, pp. 2510-2515.

[11] W. Zhang and H. Tanner. Composition of Motion Description Languages. In *Hybrid Systems: Computation and Control*, M. Egerstedt and B. Mishra (eds), Springer 2008 (to appear).

[12] J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2001.

[13] http://playerstage.sourceforge.net