Motion Programs for Multi-Agent Control: From Specification to Execution

Patrick Martin and Magnus Egerstedt

***Abstract*— This paper explores the process of turning high-level motion programs into executable control code for multi-agent systems. Specifically, we use a modified Motion Description Language (MDL) for networked systems that can specify motion programs for a collection of autonomous agents. This MDL includes the network information dependencies required for each agent to perform coordinated behaviors. We discuss the design of this framework and the language theoretic tools used to analyze the information dependencies specified by these multi-agent motion programs. Additionally, we develop a supervisor system that monitors the behavior of the agents on the network, and prevents the agents from entering into states where information dependencies are violated. We demonstrate our framework using a simulated multi-robot system.**

I. Introduction

The growing use of multi-agent robotics in diverse applications, from emergency response to warehouse supply management (e.g. [10],[16]), has created a need for efficient mission specification and control code generation for teams of autonomous agents. Using a top-down perspective, we can construct global missions for these agents using pre-made control laws, and then distribute the generated controllers to the agents.

We approach the mission specification problem by modifying the MDL framework in a way that facilitates the construction of motion programs for multi-agent systems. This new MDL, called *MDLn* (where the *n* stands for "networked"), encodes the control laws *and* the desired network information dependencies, as originally proposed in [12]. In this paper, we extend our previous results in this area and the contributions of this paper are two-fold. First, we develop a "compiler" of multi-agent motion programs that inspects the specified network dependencies among the agents and reports the existence of inconsistent network topology configurations. Furthermore, we create a tool that automatically generates an MDLn supervisor, which observes the execution of each agents' MDLn programs and prevents the system from entering into inconsistent configurations.

Other prior work in top-down specification for multi-agent systems has made use of embedded graph grammars (EGG) [11], [14]. EGGs are easy to use when the network consists 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 typically the case with self assembly systems [11], [14]. Recently, linear temporal logic methods (e.g. [15]) have been applied to specify and verify control specifications for multi-agent systems [6]. This work abstracts the dynamics of the agents and then uses LTL formulas to verify that the control specification will succeed. However, these LTL methods, in general, work best in known environments and have computational limitations associated with them.

Alternatively, we focus on systems in which the agents are heterogeneous, the environments are dynamic, and the interaction topologies may be specified *a priori*. An example of such an application is unmanned convoy protection, where a UAV is specified as the *leader* of a convoy, and a UGV convoy is tasked to follow the leader. The UAV must lead the convoy to a goal location and it must detect threats that appear in the environment. In this scenario, it may be advantageous for a human operator to specify network topologies based on the dynamic changes in the environment.

Top-down mission specification is not the only approach for controlling multi-agent robots, as shown by recent work in dynamic task allocation, e.g. [4], [7], [9]. These methods produce an emergent global behavior based on the bottom-up decision processes of the agents. Typically, the agents use distributed algorithms to select individual control tasks or decide which group of agents are best suited for information sharing. Our approach, as well as EGGs and LTL, puts more control in the hands of the system designer, which may be required when the mission specification for a multi-agent system requires a human in-the-loop.

A. Motion Description Languages

Before describing the MDLn language and framework, we need to discuss the "standard" MDL formulation. The work of [1], [8] describe MDLs as languages for composing large motion programs from collections of pre-defined controllers. We pair these controllers with an interrupt function that causes transitions between the controllers at execution.

Let the dynamic equations of a robot agent have the form

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

where x is the state, y is the sensor output, and u is the control input. This control input is defined by a mapping $\kappa : \mathcal{Y} \to \mathcal{U}$. Additionally, the interrupt function is defined as the mapping $\xi : \mathcal{Y} \to \{0, 1\}$, where 1 indicates that an interrupt has occurred. We denote a MDL mode by the tuple (κ, ξ), which means that system (1) executes the controller κ until the interrupt triggers, denoted by $\xi \to 1$.

This work was supported by the U.S. National Science Foundation for its support through grant number 0820004.

P. Martin and M. Egerstedt are with the School of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA 30332 USA patrick.martin@gatech.edu, magnus@ece.gatech.edu

MDLs allow the natural construction of motion programs for robotic systems. However, standard MDL does not encapsulate network dependencies, common in multi-agent robotic systems. We address this problem by designing a framework for constructing motion programs for networked systems based on our networked MDL language, *MDLn*, originally proposed in [12].

In Section II, we review the MDLn model and language construction. In Sections III and IV we develop the first of our contributions: compiling MDLn programs using techniques from the discrete event systems (DES) literature. Furthermore, Section V describes our second contribution: the design of our supervisor system. Our simulation results are demonstrated in Section VI

II. MDL for Multi-Agent Systems

What makes multi-agent robot mission specification conceptually different from the single robot case is that coordination and information sharing play a key role. As discussed in [12], we capture these features by modifying the dynamics of (1) to allow for the transmission of information among a collection of N agents, with index set $\mathcal{N} = \{i \in \mathbb{N} | i \leq N\}$:

$$\begin{aligned} \dot{x}^i &= f^i(x^i, u^i),\ x^i \in \mathcal{X}^i \subseteq \mathbb{R}^n, u^i \in \mathcal{U}^i \\ y^i &= h^i(x^i),\ y^i \in \mathcal{Y}^i \subseteq \mathbb{R}^p \\ s^i &= g^i(x^i, y^i), s^i \in \mathcal{S}^i \subseteq \mathbb{R}^q, \end{aligned} \tag{2}$$

where $q \leqslant \dim(\mathcal{X}^i) + \dim(\mathcal{Y}^i)$. The way these entities should be understood is as follows: agent-i's dynamics are driven by its state, x^i, under the controller, u^i. The state, x^i, determines the local information produced by the agent's sensors, y^i, and, consequently, agent-i transmits its *shareable information*, s^i, by mapping its state and sensor output onto $\mathcal{S}^i$ via the function $g^i : \mathcal{X}^i \times \mathcal{Y}^i \to \mathcal{S}^i$. [1] This information may then be transmitted through the network to a desired neighbor.

Using the model (2) we construct a multi-agent MDL, or *MDLn*, by coupling controller and interrupt functions and adding a new element for specifying the desired network information dependencies that the motion requires. This collection of dependencies, called the *buddy list*, is a collection of the neighbors with which a particular agent wants to perform actions or share information.

Explicitly, we assume that each agent has an "egocentric" network (denoted $\mathcal{W}^i$) of agents within its communication range. We define the *desired buddies* of agent-i as the set $\beta_d^i \subseteq 2^{\mathcal{N}}$, which is interpreted as a fixed set of agents with which agent-i *prefers* to communicate.

The total set of agent-i's available buddies is dependent on the current available list of agents on the network at a particular time, t, i.e.

$$\beta^i(t) = \beta_d^i \cap \mathcal{W}^i(t) \tag{3}$$

From the model in equation (2) we know that each agent chooses to share their information with the vector s^i. Assume that agent-i has k buddies, i.e. $\beta^i = \{a^1, \cdots, a^k\}$, and each of these agents transmits their information vectors: $s^1, \cdots, s^k$. Agent-i combines these vectors into a *locally* held vector denoted by $\hat{s}^i = [{s^1}^T \ \cdots \ {s^k}^T]^T \in \hat{\mathcal{S}}^i \subseteq \mathbb{R}^{kq}$. Then, agent-$i$ uses the shared information of its buddies agents when making control and interrupt decisions.

Using all of the above definitions, the control and interrupt functions from MDL are modified as follows. The control depends on the state and sensor feedback of agent-i, the information from all buddies of agent-i, and time ($\mathbb{R}^+$):

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

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 define an *MDLn mode* as the tuple $(a^i, \kappa^i, \xi^i, \beta^i)$, composed of an agent identifier, a^i, a control law, κ^i, a transition function, ξ^i, and a set of agent buddies, β^i, according to the preceding definitions. Furthermore, we define the symbol representing the k^{th} MDLn mode of agent a^i as $\sigma_k^i := (a^i, \kappa_k^i, \xi_k^i, \beta_k^i)$. The *MDLn language* is the set of all possible concatenations of these MDLn modes.

A. Agent Interaction Rules

Many multi-agent systems require that the agents be assigned different *roles*, which in turn affect the type of actions the agents can perform or information they can obtain. In addition to the convoy protection task discussed in Section I, one can imagine other leader-follower or team-based applications where agents partition the network into different command hierarchies. We use roles to specify the network hierarchy of the agents involved. Specifically, we define a role as a static value resulting from the mapping $r : \mathcal{N} \to \mathcal{R}$, where $\mathcal{R}$ is a set with total order. Agents use these roles to determine the members of the network with which they can exchange information via the following rules:

- R1: if $r(i) > r(j)$ then a^i is able to receive shared information from a^j
- R2: if $r(i) = r(j)$ then a^i and a^j may share information with each other.
- R3: if $r(i) > r(j)$ and $a^j \in \beta^i$ then a^i and a^j may share information with each other.

We interpret these rules as follows: R1 states that if the value of a^i's role is higher than the role value of a^j then a^i may pull any shareable information from a^j without restriction. R2 describes the case when a^i and a^j share the same role value, and hence can exchange their shareable information without restriction. Finally, R3 provides an exceptional case where a^j has a lower role than a^i; however, a^i already plans to work with a^j since a^j is listed in its own buddy list, β^i.

B. MDLn Example

To make the MDLn language definition more concrete, we consider an example MDLn string involving three agents, a^1, a^2, and a^3 with role assignments $r(a^1) = 1$ and $r(a^2) = r(a^3) = 0$. In this configuration, a^1 has a higher role and can

[1] Note that this product of state and output spaces may not be needed; however, the inclusion of $\mathcal{Y}^i$ makes the environmental dependence of shared information more explicit.

be considered the "leader" of a^2 and a^3. We let the agents use the controllers defined in Section I-A and add an additional controller: κ_f = `Follow`. Also, the agents are equipped with the obstacle interrupt function from Section I-A: ξ_{obs}. One example of an MDLn string using these controllers and interrupts is:

$$(a^2, \kappa_{gtg}, \xi_{obs}, \{\})(a^1, \kappa_f, \xi_{obs}, \{a^2\})(a^3, \kappa_f, \xi_{obs}, \{a^1\}). \tag{4}$$

This string tells a^2 to head toward a fixed goal location until it detects an obstacle, and consequently terminates operation. The second mode in the program instructs a^1 to follow a^2, since its buddy list is $\beta^1 = \{a^2\}$. Additionally, the third mode directs a^3 to follow a^1 due to its list, $\beta^3 = \{a^1\}$.

Agent-1 is able to execute its κ_f controller since its role is valued higher than agent-2 and is granted access to the information according to interaction rule R1. Unfortunately, agent-3 will not be able to follow agent-1 since its network dependency violates R1. This simple string reveals we need an MDLn "compiler" that can not only parse the high-level MDLn language, but also determine whether an MDLn program can be executed correctly.

III. Multi-Agent Motion Programs

The MDLn language presented in Section II provides a method for us to specify a string of motions for a set of agents. However, the strings alone do not provide enough information for determining whether the agents can execute their given programs. To do so, we need to construct a grammar that combines the role specifications defined in Section II-A with the MDLn language. A parser based on this grammar outputs MDLn strings, which each implement the controllers described by the motion program, and the associated role information of the agents. This section introduces the parser and gives an example of an acceptable MDLn program.

A. Parser

We design our *MDLn grammar* such that it can process agent role specifications in addition to the MDLn mode strings. According to the standard definition from [5], a grammar is defined by the tuple

$$G = (L, T, \Pi, \omega).$$

L represents the *non-terminals*, which are symbols that can be broken down into smaller components that are combinations of non-terminals and *terminals*, T. These terminal symbols are the smallest elements of the grammar and cannot be reduced into any other element. Π is a set of production rules that describe how the elements of L and T are composed. Finally, the grammar requires a starting symbol, which we denote as ω.

Our grammar is structured using the following production rules, Π:

$$\begin{aligned} O &\rightarrow R^\star \ M^+ \\ R &\rightarrow I \ = \ r \\ M &\rightarrow (I \ k \ z \ \{I^\star\}). \end{aligned} \tag{5}$$

Here, the nonterminals are $L = \{O, R, I, M\}$ and the terminals are $T = \{r, k, z, b, (,), =\}$. The first line of (5) shows the start symbol, $\omega = O$, which is the basic program production rule. It requires the concatenation of the symbols R and M, which represent the role assignments and modes, respectively. Note that R has a Kleene-closure operator, denoted '$\star$,' which means that our programs expect zero or more role assignments; additionally, the '$+$' operator requires that the program have at least one mode.

The R production, which creates role assignments, takes the "identifier" non-terminal I, which is similar to a variable name in standard programming languages, followed by the helper terminal, $=$, and role map terminal, r. For example, the R productions described for the MDLn string in (4) are written as:

```
a1 = 1
a2 = 0
a3 = 0
```

The mode production, M, is made by concatenating the nonterminal I (also an identifier) with terminals k and x and a string of identifier symbols, $I^\star$. Using this production we can write the mode string corresponding to (4) as:

```
(a2, GoToGoal, Obstacle, {})
(a1, Follow, Obstacle, {a2})
(a3, Follow, Obstacle, {a1})
```

In this code snippet, the controllers from the grammar, k, are represented by the symbols `GoToGoal` and `Follow`. Additionally, the z element from the mode production is represented by the three `Obstacle` symbols. The buddy lists for each agent are the identifiers listed between the `{}` elements. This grammar allows us to create MDLn programs that are *syntactically* correct; however, we still need to design a process that ensures that MDLn programs can execute correctly when deployed onto the networked agents.

IV. MDLn Program Consistency

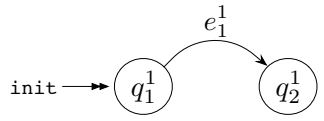
Before deploying an MDLn program onto a team of mobile robots, we must check whether the program correctly specifies the network information dependencies of the agents, which are represented by the buddy lists within each MDLn mode. We construct this consistency checker based on techniques from discrete event systems (DES) and language theory (e.g. [2], [5], [13]). Since MDLn strings specify a sequence of controllers, we construct an automaton representing the sequential execution of the system.

This automaton is defined as the tuple

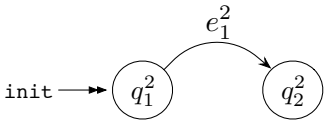
$$A^i = (Q^i, E^i, \delta, q_0^i, o^i),$$

where Q^i is the set of states representing agent-i's modes within its MDLn string of length m. Also, E^i is the set of events corresponding to each of the interrupt functions in the MDLn string. The transition function, $\delta : Q^i \times E^i \rightarrow Q^i$, defines the transition to different states of A from the initial state, q_0. Each state, $q_k^i \in Q^i$, represents the the execution of the controller κ_k^i by agent a^i. The set E^i is made up

of the transitions from one MDLn mode to another caused by the interrupt $\xi_k^i \rightarrow 1$. Additionally, the automata have an output function, $o^i : Q^i \rightarrow 2^{\mathcal{N}}$, that maps each state to the buddy list of that current MDLn mode, i.e. for some q_k^i, $o(q_k^i) = \beta_k^i \in 2^{\mathcal{N}}$, where $\mathcal{N}$ is the agent index set defined in Section II.



(a) A^1

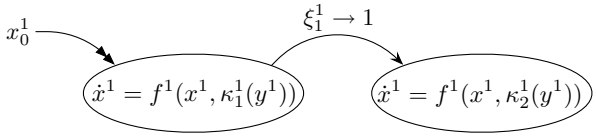


(b) A^2

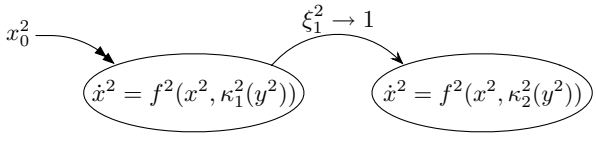
Fig. 1. Two automata representing the MDLn strings for agent-1, 1(a), and agent-2, 1(b).

For example, consider the MDLn string: $\sigma_1^1\sigma_2^1\sigma_1^2\sigma_2^2$. Figure 1 shows the constructed automata that represents agent-1 and agent-2 executing their given MDLn modes: $\sigma_1^1\sigma_2^1$ and $\sigma_1^2\sigma_2^2$, respectively. Additionally, Figure 2 illustrates the hybrid dynamics of the systems using the same MDLn modes.

Automaton, A^1, in Figure 1(a) starts out running in state q_1^1, which has the output map $o(q_1^1) = \beta_1^1$. While in the state q_1^1, the dynamics of the system are taken from the first state in Figure 2(a): $\dot{x}^1 = f^1(x^1, \kappa_1^1(y^1))$. Once this mode is interrupted by $\xi_1^1 \rightarrow 1$, the dynamics are changed to $\dot{x}^1 = f^1(x^1, \kappa_2^1(y^1))$ and the event e_1^1 causes a transition in A^1 to its next state, q_2^1. The automaton, A^2, executes in a similar manner; however, its controllers and interrupts are independent of A^1's.



(a) Dynamics of agent-1.



(b) Dynamics of agent-2.

Fig. 2. The hybrid automata representing the dynamics of the two robots as they execute their given MDLn strings.

These automata adequately represent the *individual* behavior of each agent; however, the MDLn language is designed to specify tasks for a collection of agents. Therefore, we use these automata to analyze how the networked information dependencies of each agent affect the consistency of the MDLn program. A natural operation for analyzing these automata is the *parallel composition*. By composing these automata, we see how the execution of each agent's string affects all of the agents' behavior during execution of their individual MDLn strings.

Consider, again, the automata in Figure 1. Using parallel composition, we generate a so called *network automaton*, shown in Figure 3, and denote it as $\mathcal{A}^{12} = A^1 \| A^2$. This automaton has four states and two possible events, which are taken from the event sets of the individual automata, i.e. $E = E^1 \cup E^2 = \{e_1^1, e_1^2\}$. This network automaton represents the global behavior of the agents as each executes its own MDLn string.

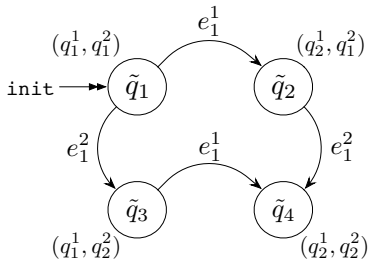


Fig. 3. The network automaton, $\mathcal{A}^{12}$, generated by the composition of the automata in Figure 1. The states from automata A^1 and A^2 are written next to the states of $\mathcal{A}^{12}$ to explicitly show the states included in the network automaton.

A. Network Automata

We use network automata, which were mentioned in the prior section, to determine when an MDLn program is inconsistent. To do so, any state in the network automaton with incorrect information dependencies should be *marked*. Then, the goal of the consistency checker is to determine whether there are languages over the automaton that lead to marked states.

More precisely, assume we have n agents, and each agent has a finite number of modes and k_i denotes the k^{th} mode of a^i. Each automata is constructed from the agents' associated MDLn program. These automata are then composed into the network automaton, $\mathcal{A} = A^1 \| \cdots \| A^n$. This network automaton is defined by the tuple:

$$\mathcal{A} = (\mathcal{Q}, \mathcal{E}, \tilde{\delta}, q_0, \mathcal{Q}_m).$$

Each state, $\tilde{q} \in \mathcal{Q} = Q^1 \times \cdots \times Q^n$, combines the current state from each agent in the current MDLn program. For n agents, a state is defined by $\tilde{q}_l = (q_{k_1}^1, \cdots, q_{k_n}^n)$, where l indexes the states of $\mathcal{Q}$. Returning to the two-agent example in Figure 3, the state state $\tilde{q}_1$ represents the pair of states in Figures 1(a) and 1(b): (q_1^1, q_1^2).

The event set, $\mathcal{E}$, is the union of the individual events from all event sets from each agent's automaton generated by their

MDLn string, i.e. $\mathcal{E} = E^1 \cup \cdots \cup E^n$. Additionally, a new mapping, $\tilde{\delta} : \mathcal{Q} \times \mathcal{E} \to \mathcal{Q}$, transitions among the states in $\mathcal{Q}$. Finally, a subset of the automaton's states, $\mathcal{Q}_m \subseteq \mathcal{Q}$, are marked as *inconsistent*.

B. Inconsistent States

The consistency of a state in the network automaton is determined by the agents' networked information dependencies during the execution of their current modes. The MDLn program that generated the individual automata in Figure 1 may have roles assigned to each agent. Using these roles, we mark states of Figure 3 if the rules discussed Section II-A are violated. For example, we can construct a *pairwise* logic predicate that indicates if a state from the two-agent network automaton in Figure 3 should be marked:

$$(q^1_{k_1}, q^2_{k_2}) \in \mathcal{Q}_m \Leftrightarrow P(q^1_{k_1}, q^2_{k_2}) \tag{6}$$

where,

$$\begin{aligned} & P(q^1_{k_1}, q^2_{k_2}) := \\ & (a^1 \notin o(q^2_{k_2}) \wedge a^2 \in o(q^1_{k_1}) \wedge r(a^1) < r(a^2)) \\ \vee \quad & (a^2 \notin o(q^2_{k_2}) \wedge a^1 \in o(q^2_{k_2}) \wedge r(a^2) < r(a^2)), \end{aligned} \tag{7}$$

where $\vee$ and $\wedge$ denote logical disjunction and conjunction, respectively. The logical statement expressed by equation (6) means that a state from $\mathcal{A}^{12}$ is inconsistent if and only if a^1 is *not* in a^2's buddy list *and* a^1 depends on a^2 *and* a^1's role value is less than a^2's value; or, *vice versa*.

To generalize equation (6) to the case of n-agents, we take the disjunction over all possible pairs that are part of state, $\tilde{q}_l \in \mathcal{Q}$. (If the pair $(q^i_{k_i}, q^j_{k_j})$ is part of the state $\tilde{q}_l$, we denote the relationship by the symbol '$\subset$'.) In other words, using equation (7), a state in $\mathcal{Q}$ is inconsistent if the following holds:

$$\tilde{q}_l \in \mathcal{Q}_m \Leftrightarrow \bigvee_{(q^i_{k_i}, q^j_{k_j}) \subset \tilde{q}_l} P(q^i_{k_i}, q^j_{k_j}). \tag{8}$$

Returning to the two-agent example network automaton in Figure 3, assume that the particular MDLn program sets the role values as $r(a^1) < r(a^2)$. Additionally, let a^1 depend on a^2 while executing its second mode, i.e. $o^1(q^1_2) = \{a^2\}$; however, a^2 operates independently while executing its first mode: $o^2(q^2_1) = \emptyset$. Applying equation (8) to each state in $\mathcal{Q}$ results in the marking of $\tilde{q}_2$, shown in Figure 4, since $a^1 \notin o^2(q^2_1) = \emptyset$, $a^2 \in o^1(q^1_2)$, and a^1's role value is lower than a^2's value.

An automaton with marked states generates a *marked* language, or set of strings, that enumerates the behaviors of the system leading to inconsistent states. The marked language of the network automaton in Figure 4 is $\mathcal{L}_m(\mathcal{A}^{12}) = \{e^1_1\}$, since e^1_1 is the only event that leads to $\tilde{q}_2$. In the following section we will see how this marked language can be used to develop a supervisor that disables events leading to inconsistent states, such as $\tilde{q}_2$.

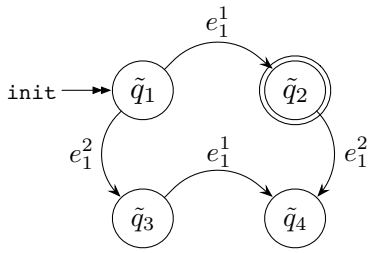


Fig. 4. The two-agent network automaton, $\mathcal{A}^{12}$, with state $\tilde{q}_2$ marked as inconsistent.

V. Supervisors for Multi-Agent Motion Programs

Section IV developed a way to model MDLn program specification and indicate when an MDLn program will not execute correctly because of inconsistent states. In order to prevent the transition to bad states in $\mathcal{Q}$, we propose a *supervisor* that blocks events leading to inconsistent states.

More formally, the supervisor is an automaton represented by the tuple

$$\mathcal{S} = (W, \mathcal{E}, \lambda, w_0, \phi)$$

where W is the set of states, with w_0 as the initial state, and $\mathcal{E}$ is the same set of events from $\mathcal{A}$. The mapping $\lambda : W \times \mathcal{E} \to W$ provides the transition dynamics among the states in W. This supervisor accepts event strings from $\mathcal{A}$ and outputs some control signal, γ.

These controls are symbols from a subset of the possible events in the network automaton: $\gamma \in \Gamma \subseteq \mathcal{E}$. Consequently, we define the mapping $\phi : W \to \Gamma$ that maps the currently active supervisor state to a control input, $\gamma \in \Gamma$. The network automaton is given the controlled subset of events, which in turn disables the events leading to inconsistent states.

Note that the automaton $\mathcal{A}^{12}$ in Figure 4 will enter into the marked state $\tilde{q}_2$ if the event e^1_1 is taken from state $\tilde{q}_1$. To prevent this behavior from occurring, we construct $\mathcal{S}$ by exploring the network automaton with a depth-first-search (DFS) algorithm [3], adding states to the supervisor and augmenting the function ϕ at each state.

The supervisor for $\mathcal{A}^{12}$ in Figure 4 is generated in the following way. We start at $\tilde{q}_1$ in $\mathcal{A}^{12}$ and create an initial state in $\mathcal{S}$, $w_0 := w_1$, as shown in Figure 5. At this state, the current event string is simply the empty string, ϵ, and the control function is initialized as $\phi(w_1) = \{e^1_1, e^2_1\}$. We then examine the available transitions out of state $\tilde{q}_1$. By taking e^1_1, we encounter the inconsistent state $\tilde{q}_2$ *and* the current string e^1_1 is an element of $\mathcal{L}_m(\mathcal{A}^{12})$. Therefore, this event must be disabled at state w_1 by excluding it from the control function: $\phi(w_1) = \{e^2_1\}$. Next, we return to state $\tilde{q}_1$ (via DFS) and take the last remaining event to $\tilde{q}_3$, which is a consistent state. We create the next state, w_2, in the supervisor and connect it to w_1 with event e^2_1; additionally, the control function at w_2 is initialized as $\phi(w_2) = e^1_1$. After making the final jump to state $\tilde{q}_4$ with e^1_1, we see that there are no more states in $\mathcal{Q}$ and $\tilde{q}_4$ is not an inconsistent state;

therefore, the process completes by adding with a final state w_3 to the supervisor and setting $\phi(w_3) = \emptyset$.

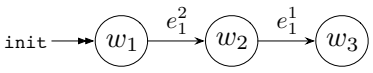


Fig. 5. The constructed supervisor for the network automaton $\mathcal{A}^{12}$ in Figure 4.

A. MDLn Supervisor Deployment

In summary, the MDLn compilation process involves the inconsistency analysis of Section IV and the *automatic* generation of a supervisor automaton. If a MDLn program is created such that its initial state is inconsistent, then a supervisor automaton will not be generated. Otherwise, the supervisor automaton is created and is deployed within a *supervisor agent* that can monitor the other MDLn agents on the network.

This supervisor agent inspects the current state of its automaton and applies the $\phi(\cdot)$ function. If any events should be disabled on the network, the supervisor issues `hold` messages to those agents with the disabled events. Additionally, as the MDLn agents execute their programs, they transmit `transition` messages that cause the supervisor to advance its automaton and setup the next set of enabled events. Examples of this implementation are described in the next section.

VI. Simulation Results

In this section, we demonstrate the simulation of the MDLn framework discussed through Sections II–V using the robotics simulation environment *Player/Stage*[2] as our back-end. We use our multi-agent software infrastructure, *Pancakes*, to create and manage the agents on the network.

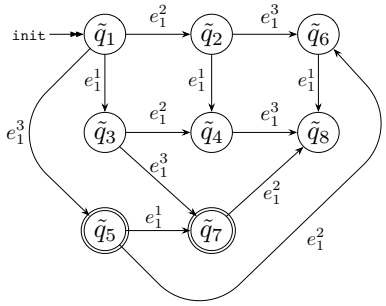
A. Example: Basic Supervisor Operation

Assume we are given two agents that share a leader role and a single follower, i.e. $r(a^1) = r(a^2) > r(a^3)$. The MDLn program is:

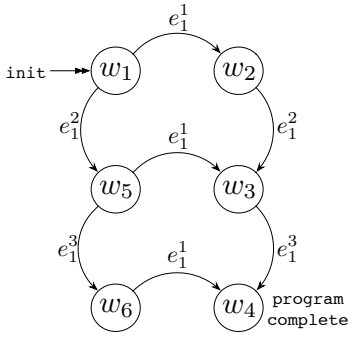
```
a1 = 1, a2 = 0, a3 = 0
(a1, GoToGoal, AtGoal, {a3})
(a1, Wait, Always, {a3})
(a2, GoToGoal, AtGoal, {})
(a2, Wait, Always, {a3})
(a3, Follow, BuddyAtGoal, {a1})
(a3, Follow, Obstacle, {a2})
```

This MDLn program instructs a^1 to approach the goal and then, once the goal is reached, wait indefinitely; additionally, a^1 is instructed to share its information with a^3. A similar string is given to a^2, but it excludes a^3 from knowing its information until it switches to its second mode: `(a2, Wait, Always, {a3})`. Finally, a^3 is instructed to follow a^1 until a^1 reaches the goal, and then switch to following a^2 until it detects an obstacle. Note that a^3 uses the `BuddyAtGoal` interrupt for determining when it should switch to following a^2. Since `BuddyAtGoal` uses the shared (and possibly noisy) information of a^1, it is possible for the interrupt to fire before a^1 actually reaches the goal.

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



(a) The network automaton generated by the example MDLn program.



(b) The supervisor generated from the follow-the-leader MDLn program.

Fig. 6. The network and supervisor automata generated by the compilation of the MDLn example program.

The network automaton generated by the compilation of this program is shown in Figure 6(a). Note that if a^3 switches to its second mode it creates an inconsistent state since it depends on the information from a^2 to execute the `Follow` controller. Additionally, the network dependencies are still inconsistent if a^1 switches to its second mode and a^3 still attempts to follow a^2. Therefore, by applying the logic predicate from equation (8), states $\tilde{q}_5$ and $\tilde{q}_7$ are marked as inconsistent.

The compilation automatically generates the supervisor shown in Figure 6(b). This supervisor automaton is deployed inside the supervisor agent, denoted SA, which receives messages from the MDLn agents, a^1, a^2, and a^3, every time a transition in the MDLn program occurs. When the program starts, the supervisor starts in state w_1 with only two enabled events e_1^1, e_1^2. The SA issues a `hold` message to a^3, which prevents the a^3 from executing its second mode (Figure 7). Once a^1 reaches the goal, the event e_1^1 is transmitted to the SA and the supervisor advances to state w_2. In this state, a^3 is still held from advancing its program. After a^2 reaches the

goal, the event e_1^2 advances the supervisor to state w_3 and a `release` message is sent to a^3. Finally, a^3 executes its `Follow` behavior and completes the program once it reaches a^2, as shown in Figure 8.

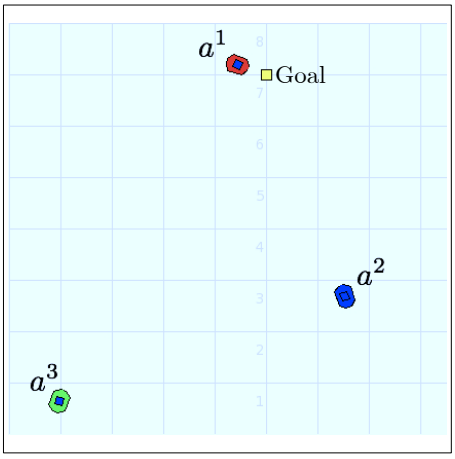


Fig. 7. This image shows a^3 being held after initially following a^1.

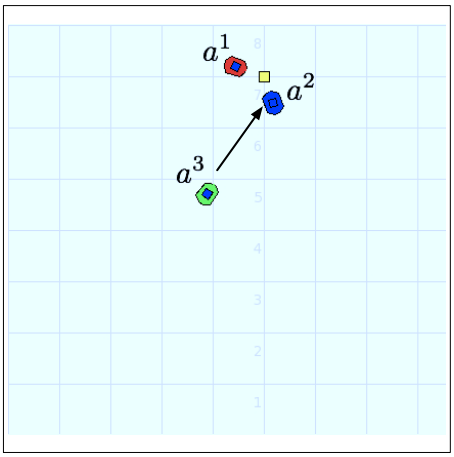


Fig. 8. Once a^2 reaches the goal the SA releases a^3 and a^3 follows a^2 to the goal.

B. *Example: Threat Detection*

In this example, we construct a more complicated MDLn motion program that uses four heterogeneous agents. We partition the network using the following role values: $r(a^1) = 2, r(a^2) = r(a^3) = 1, r(a^4) = 0$. The a^1 agent has a sensor that can identify possible threats to the other agents. Two other agents, a^2 and a^3, are tasked with exploring the environment and a^4 is set to follow a^3 for exploration redundancy.

The MDLn program scripted for this example is:

```
(a1, GoToGoal, ThreatDetected, {a2 a3})
(a1, ApproachThreat, AtThreat, {})
(a1, ScanThreat, 10, {})
(a1, GoToGoal, AtGoal, {a2 a3})
(a2, Explore, Obstacle or 10, {})
(a2, Follow, Obstacle, {a1})
(a2, Avoid, Clear, {})
(a3, Explore, Obstacle or 10, {a4})
(a3, Follow, Obstacle, {a1})
(a3, Avoid, Clear, {})
(a4, Follow, Obstacle, {a3})
(a4, Avoid, Clear, {})
(a4, GoToGoal, AtGoal, {})
```

This more complicated program has a^1 go to the goal until it detects a threat, all the while sharing information with a^2 and a^3. When it detects the threat, it must approach and perform a `ScanThreat` behavior for 10 time units. After threat scanning is complete, a^1 is instructed to continue towards the goal. The strings for a^2 and a^3 are similar: they both explore until they see an obstacle *or* 10 time units have passed. Once that is complete they both follow a^1 until an obstacle is detected. Finally, a^4 follows a^3, which has $a^4 \in \beta^3$, until a^4 detects an obstacle. If that event occurs, a^4 continues to the goal position.

The network automata generated by this program has 108 possible states, 46 of which are inconsistent, and the resulting supervisor contains 62 states. We deploy this program onto the four agents shown in Figure 9 and a similar supervisor agent, SA. This example program shows how much more complicated the network automata and supervisors become by adding a few modes and agents. However, it also shows that our automatic tool is useful for the analysis of the potentially complicated network interactions specified by an MDLn program.

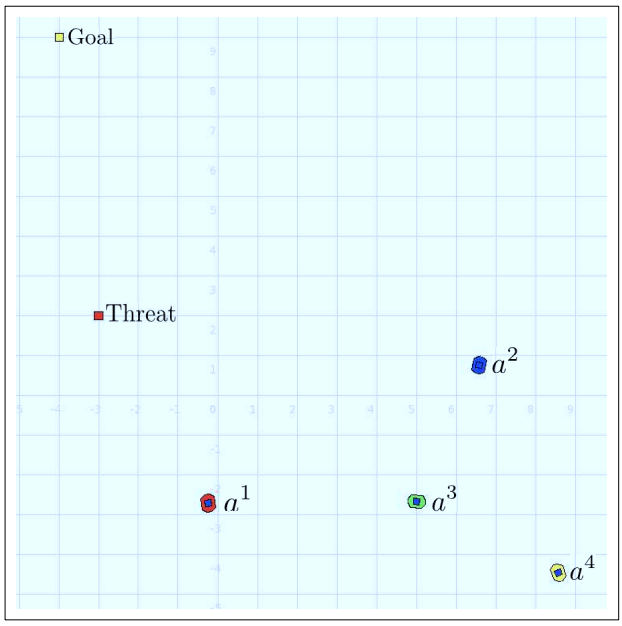


Fig. 9. The initial configuration of the four agents for the threat detection example.

Initially, a^1 detects and approaches the target in Figure

10(a). At this point in the program, a^2 and a^3 cannot receive a^1's position information for following, since a^1 is scanning the threat by itself. The SA holds a^2 and a^3 to prevent the entry into an inconsistent state; however, a^4 can still follow a^3. Once a^1 finishes scanning the threat, it proceeds towards the goal and again a^2 and a^3 are allowed to get a^1's shared information. The event e_3^1 of a^1's MDLn string fires and advances the supervisor automaton in SA, which re-enables a^2 and a^3. Figure 10(b) shows the completion of the program, as a^1 reaches the goal with a^2 and a^3 following and a^4 executing its `GoToGoal` behavior.

VII. Conclusion

In this paper, we presented a framework based on an extended MDL for specifying high-level tasks for a collection of agents. We also discussed the ability of this framework to compile MDLn programs, which indicate to users whether the composed motion program is valid for the specified network topologies. Finally, we demonstrated this framework by composing, compiling, and executing MDLn programs on a set of simulated robots.

Acknowledgement

The authors thank Jean-Pierre de la Croix for helpful interactions.

References

[1] R.W. Brockett. On the computer control of movement. In *Proceedings of the 1988 Conference of Robotics and Automation*, pages 534-540, April 1988.

[2] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, New York, NY, 2008.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 2005.

[4] B. Gerkey and M. Mataric. A formal analysis and taxonomy of task allocation in multi-robot systems. *International Journal of Robotics Research*, 23(9):939-954, September 2004.

[5] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 2006.

[6] M. Kloetzer and C. Belta. Temporal logic planning and control of robotic swarms by hierarchical abstractions. *IEEE Transactions on Robotics*. Vol. 23, pp. 320-330, April 2007.

[7] T. Leuth and T. Laengle. Task description, decomposition, and allocation in a distributed autonomous multi-agent robot system. In *Proceedings of International Conference on Intelligent Robots and Systems*, pages 1516-1523, September 1994.

[8] 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.

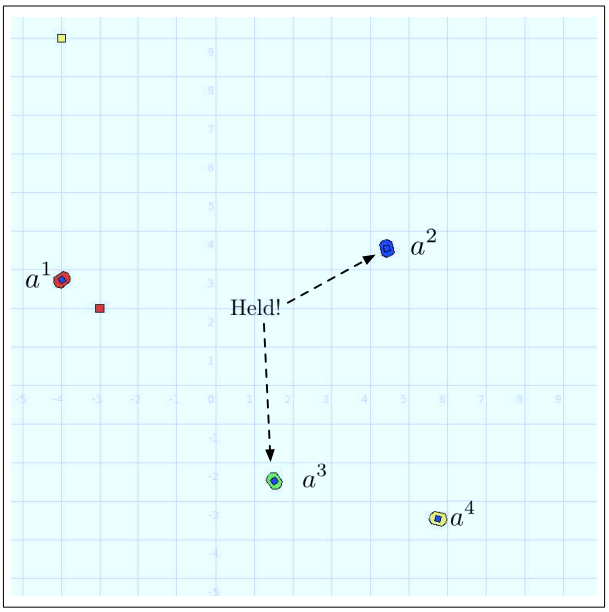
[9] J. McLurkin and D. Yamins. Dynamic task assignment in robot swarms. *Robotics: Science and Systems I*. Cambridge, Massachusetts, 2005.

[10] E.H. Ostergaard, M. Matarić, and G. Sukhatme. Distributed multi-robot task allocation for emergency handling. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, Wailea, Hawaii, October 2001.

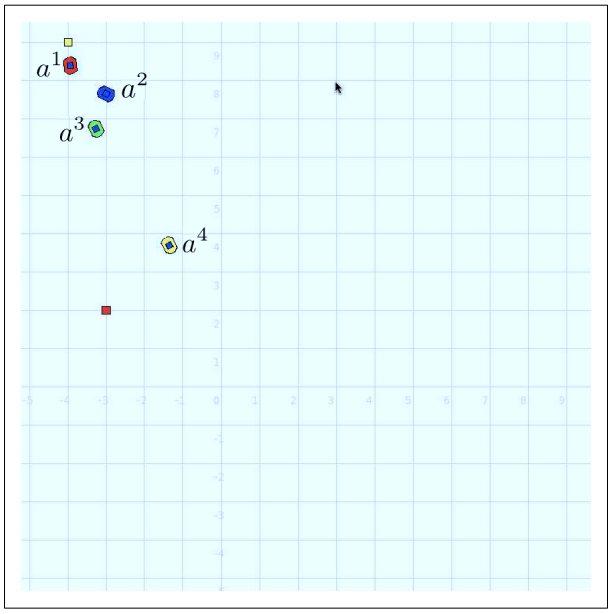
[11] J.M. McNew and E. Klavins. Locally interacting hybrid systems with embedded graph grammars. In *45th IEEE Conference on Decision and Control*. 2006, pages 6080-6087.

[12] P. Martin, J.P. de la Croix, and M. Egerstedt. MDLn: A motion description language for networked systems. In *Proceedings of 47th IEEE Conference on Decision and Control*, pages 558-563, December 2008.

[13] P.J.G. Ramadge and W.M. Wonham. The Control of Discrete Event Systems. *Proceedings of the IEEE*, Vol. 77, No. 1, pages 81-98, January 1989.



(a)



(b)

Fig. 10. Images of the simulation running the example MDLn program. The image in Figure 10(a) shows a^2 and a^3 being held by the SA while a^1 scans the target. Figure 10(b) shows the final completion of the simulation with all agents approaching the final destination.

[14] B. Smith, A. Howard, J.M. McNew, J. Wang, M. Egerstedt. Multi-robot deployment and coordination with embedded graph grammars. Journal of Autonomous Robots, 26:79-98, Springer 2009.

[15] P. Tabuada and G.J. Pappas. Model checking LTL over controllable linear systems is decidable. *Lecture Notes in Computer Science*, O. Maler and A. Pnueli, Eds. Springer-Verlag, 2003, vol. 2623.

[16] P. Wurman, R. D'Andrea, and M. Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine*, 29(1):1-19, 2008.