The MDM Package: Software Description and Deployment Guidelines

João Vicente Messias

June 2013

1 Introduction

This document describes the "Markov Decision Making" (MDM) software package and its applications. MDM is being developed in the context of MAIS+S project (CMU-PT/SIA/0023/2009) with the goal of supporting the deployment of decision-making methodologies based on Markov Decision Processes (MDPs) to teams of physical autonomous agents.

Decision-theoretic methods, such as MDPs and their related extensions, provide a solid theoretical basis for decision making under uncertainty. These frameworks, and their application to real-world multiagent surveillance environments, constitute one of the central research topics of the MAIS+S project. In practice, the goal of these methods is to obtain a *plan* or *policy*, which specifies the *actions* that an autonomous agent (or a group of such agents) should take, while considering some measure of uncertainty in the environment. However, modeling a real-world system as an MDP (or as a Partially Observable MDP – a POMDP) typically requires the abstraction of relevant characteristics of the environment. Consequently, a plan which results from such an MDP also provides symbolic, abstract actions, which are not directly applicable to the *control inputs* of an autonomous agent. Therefore, agents must be provided with some means of interpreting the environment through the scope of its associated MDP, and must also be capable of interpreting the actions that the solution to that MDP provides as concrete references to its actuators. This problem is depicted in Figure 1.

In most cases, both sides of this abstraction are carried out in an *ad-hoc* approach, tailored to the task at hand and the particular policy that a given MDP is supposed to provide. This results, however, in a large amount of

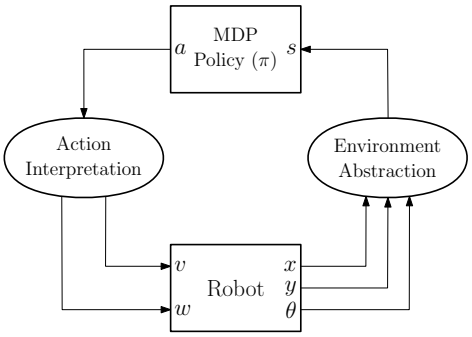

Figure 1: An example of the integration of an MDP-based control policy to a robotic agent. The actual control inputs of the robot are its linear and angular velocities (v and w resp.). It is able to sense its location in the environment (the triplet $\langle x, y, \theta \rangle$). The MDP control policy requires the abstraction of this information as a symbolic "state" s. In response, it provides an adequate symbolic action a, which must be then interpreted by the robot as a sequence of appropriate control inputs.

implementation-specific work by the problem designer, which is difficult to reuse in other, similar applications, and is subject to designer errors during deployment, which, in turn, are difficult to track, and may invalidate the MDP model of the system and/or its solution.

The purpose of the MDM package is to provide researchers / autonomous systems engineers with an easy-to-use set of tools for the deployment of MDP-based decision making methods to physical agents. Its features include:

- The ability to easily abstract system/environment characteristics as symbolic *states* or *observations* (for discrete MDPs or POMDPs, respectively);
- Supports single agent and multiagent systems;
- Generic callback-based action interpretation allows actions to be defined through ROS-native frameworks (e.g. actionlib/SMACH);
- The ability to implement hierarchical MDPs/POMDPs;
- Supports synchronous (fixed-rate) and asynchronous (event-driven) execution strategies;
- Relevant execution information can easily be logged through ROS (actions, states, rewards, transition rates, etc.);

- MDM uses the Multiagent Decision Process (MADP) Toolbox. MADP is a toolbox for decision-theoretic research, containing state-of-the-art solution algorithms for multiagent MDP-based models, and is actively maintained and extended by researchers in that field. MDM can potentially implement any model which can be defined through MADP.

2 Terminology

This document assumes that the reader is familiar with basic ROS concepts and terminology. Fundamental concepts include ROS *topics*, *services*, *parameters* and *namespaces*). `C++` examples also assume that the reader is familiar with the implementation of simple ROS nodes in that programming language.

Furthermore, to deploy this software, or to understand its organization, the reader should be acquainted with the basic theoretical aspects of decision making under uncertainty (discrete MDPs, discrete POMDPs, and multiagent instantiations of these models).

The following notation will be used in the remainder of this document. We will define a multiagent MDP (MMDP) as a tuple $\langle d, \mathcal{S}, \mathcal{A}, T, R \rangle$, where:

d is the number of agents;

$\mathcal{S} = \mathcal{X}_1 \times \mathcal{X}_2 \times \ldots \times \mathcal{X}_k$ is the *state space*, a discrete set of possibilities for the state s of the process. The (joint) state $s \in \mathcal{S}$ is a tuple of *state factor* assignments: $s = \langle x_1, x_2, \ldots, x_k \rangle$, where $x_i \in \mathcal{X}_i$;

$\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_d$ is a set of *joint* actions. Each joint action $\mathbf{a} \in \mathcal{A}$ is a tuple of individual actions: $\mathbf{a} = \langle a_1, a_2, \ldots, a_d \rangle$, where $a_i \in \mathcal{A}_i$;

$\mathcal{O} = \mathcal{O}_1 \times \mathcal{O}_2 \times \ldots \times \mathcal{O}_d$ is the space of joint observations $\mathbf{o} = \langle o_1, ..., o_d \rangle$, where $o_i \in \mathcal{O}_i$ are the individual observations of each agent.

$T : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the transition function, such that $T(s', s, \mathbf{a}) = \Pr(s'|s, \mathbf{a})$;

$O : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the observation function, such that $O(\mathbf{o}, s', \mathbf{a}) = \Pr(\mathbf{o}|s', \mathbf{a})$;

$R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the *reward function*. The value $R(s, \mathbf{a})$ represents the "reward" that the team of agents receives for performing joint action $\mathbf{a}$ in state s;

Furthermore, we will define a multiagent POMDP (MPOMDP) as a tuple $\langle d, \mathcal{S}, \mathcal{A}, \mathcal{O}, T, O, R \rangle$, where:

$d, \mathcal{S}, \mathcal{A}, T, R$ are defined as in a MDP

$\mathcal{O} = \mathcal{O}_1 \times \mathcal{O}_2 \times \ldots \times \mathcal{O}_d$ is the space of joint observations $\mathbf{o} = \langle o_1, ..., o_d \rangle$, where $o_i \in \mathcal{O}_i$ are the individual observations of each agent.

$O : \mathcal{O} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is the observation function, such that $O(\mathbf{o}, s', \mathbf{a}) = \Pr(\mathbf{o}|s', \mathbf{a})$;

Single-agent MDPs and POMDPs are here simply seen as instantiations of MMDPs and MPOMDPs such that $d = 1$.

3 MDM Overview

At its core, MDM is organized into a set of *Layers*, which embody the components that are involved in the decision making loop of a generic MDP-based agent:

- The *System Abstraction Layer* – This is the part of MDM that constantly maps the sensorial information of an agent to the constituent abstract representations of a (PO)MDP model. This layer can be implemented, in turn, by either a *State Layer* or an *Observation Layer* (or both), depending on the observability of the system;
- The *Control Layer* – Contains the implementation of the policy of an agent, and performs essential updates after a decision is taken (e.g. belief updates);
- The *Action Layer* – Interprets (PO)MDP actions. It associates each action with a user-definable callback function.

MDM provides users with the tools to define each of these layers in a way which suits a particular decision making problem, and manages the interface between these different components within ROS.

An MDM *ensemble* is an instantiation of a System Abstraction Layer, a Control Layer, and an Action Layer which, together, functionally implement an MDP, POMDP, or any other related decision-theoretic model. A basic MDM control loop is shown in Figure 2, which embodies a single-agent MDP. The inputs and outputs to each of the various ROS components are also made explicit.

In the following subsections, the operation of each of these core components is described in greater detail.

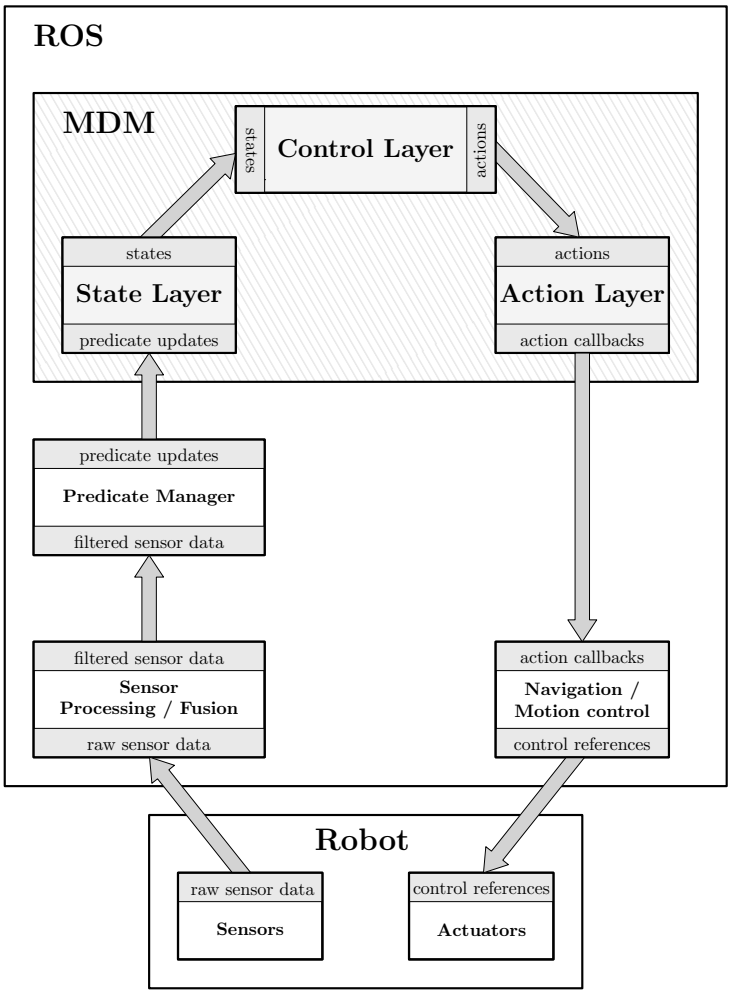

Figure 2: The control loop for an MDP-based agent using MDM.

3.1 The State Layer

From the perspective of MDM and for the rest of this document, it is assumed that a robotic agent is processing or filtering its sensor data through ROS, in order to estimate relevant characteristics of the system as a whole (*e.g.* its localization, atmospheric conditions, etc.). This information can be mapped to a space of discrete factored states by a State Layer, if and only if there is no perceptual aliasing and there aren't any unobservable state factors. The State Layer constitutes one of the two possible System Abstraction Layers in MDM (the other being the Observation Layer).

Functionally, State Layers translate sets of *logical predicates* into factored, integer-valued state representations. For this purpose, MDM makes use of

the concurrently developed *Predicate Manager* ROS package . In its essence, Predicate Manager allows the user to easily define semantics for arbitrary logical conditions, which can either be directly grounded on sensor data, or defined over other predicates through propositional calculus. Predicate Manager operates asynchronously, and outputs a set of *predicate updates* whenever one or more predicates change their logical value (since predicates can depend on each other, multiple predicates can change simultaneously).

From the perspective of ROS/MDM, predicates are seen as named logical-valued structures. More formally, let p represent a predicate and $l_t(p) \in \{0, 1\}$ represent the logical value of p at some time $t \in \mathbb{R}_0^+$. Given a set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$, and a factored state description $\mathcal{X} = \{\mathcal{X}_1 \ldots, \mathcal{X}_m\}$, the State Layer establishes a surjective map $\mathcal{H}_i : \{0, 1\}^k \to \mathcal{X}_i$ for each state factor i, that is, it maps length-k logical vectors onto each discrete set $\mathcal{X}_i$. These logical vectors, in turn, are the values of k predicates in a given subset $\mathcal{P}' = \{p'_1, \ldots, p'_k\} \subseteq \mathcal{P}$. That is, such a vector can be taken at time t as $v_t^{\mathcal{P}'} \in \{0, 1\}^k : [v_t^{\mathcal{P}'}]_i = l_t(p'_i) \, \forall i \in \{1, \ldots, k\}$.

Predicates can be mapped onto discrete *state factors* by a State Layer, in one of the following ways:

- A binary state factor can be defined by binding it directly to the value of a particular predicate. That is, for the i-th state factor and j-th predicate, $x_i|_t = l_t(p_j) \, \forall t \in \mathbb{R}_0^+$.
- An integer-valued factor can be defined by an ordered set of **mutually exclusive** predicates, under the condition that one (and only one) predicate in the set is true at any given time. The index of the true predicate in the set is taken as the integer value of the state factor. Formally, the given (sub)set of predicates $\mathcal{P}' \subseteq \mathcal{P}$ must be such that $l_t(p'_1) \vee l_t(p'_2) \vee \ldots \vee l_t(p'_k) = 1$ and $l_t(p'_u) \wedge l_t(p'_v) = 1 \implies u = v$, $\forall t \in \mathbb{R}_0^+$. Then, for each time t, $x_i|_t = \iota_t$ such that $\iota_t = \min\{k : l_t(p'_k) = 1\}$.

The State Layer always outputs the *joint* value of the state factors that it contains, *i.e.* a single integer value which univocally corresponds to an assignment of state factors. This minimizes the amount of information that needs to be passed between modules. However, for multiagent problems, note that this does not mean that the whole *joint state* of the system needs to be modeled in each State Layer for every agent. If each agent can only access *part* of the state space, for example in a Dec-MDP, then different State Layers can model different subsets of state factors.

3.1.1 Implementing a State Layer

The following example demonstrates how a State Layer ROS node can be implemented in C++, for a small example in the context of MAIS+S. There are two state factors in the problem: the first is an integer-valued state factor describing the topological location of a robot in its environment; the second is a binary variable representing whether or not there is a person waiting for assistance.

```
#include <ros/ros.h>

#include <markov_decision_making/StateLayer.h>

using namespace ros;
using namespace markov_decision_making;

int main (int argc, char** argv)
{
  init (argc, argv, "state_layer_example");

  StateLayer robot_mls;
  robot_mls.addStateFactor (StateDep()
                            .add ("IsInElevatorHallway")
                            .add ("IsInWestCorridor")
                            .add ("IsInSouthCorridor")
                            .add ("IsInCoffeeRoom")
                            .add ("IsInLRM")
                            .add ("IsInSoccerField"));
  robot_mls.addStateFactor (StateDep()
                            .add ("PersonIsWaiting"));
  spin();

  return 0;
}
```

We will now analyze the code (besides the bare minimum ROS node initialization / support):

```
StateLayer robot_mls;
```

This declares our State Layer, which will be silently connecting to Predicate Manager topics (`~/predicate_map` and `~/predicate_updates`), but will *not* be spinning its own thread. Note that the `spin()` function is still handled externally, and it is only called *after* state factor variables are declared. The State Layer will be publishing its state information to the `~/state` topic.

```
robot_mls.addStateFactor (StateDep()
                          .add ("IsInElevatorHallway")
                          .add ("IsInWestCorridor")
                          .add ("IsInSouthCorridor")
                          .add ("IsInCoffeeRoom")
                          .add ("IsInLRM")
                          .add ("IsInSoccerField"));
```

This adds an integer state factor to the State Layer, and binds its value to a set of mutually exclusive predicates which describe whether or not the robot is in a particular topological position. State factors must be added in the order that the user wants them to appear in the factored state description - that is, this code snippet will be considered as state factor $\mathcal{X}_1$. Likewise, predicates are added as dependencies in the order that they should be assigned to state factor values – `"IsInElevatorHallway"` will correspond to the first value of this state factor. The `StateDep()` class is used to easily register a chain of predicates as dependencies. Note that, following the C++ standard and to maintain consistency with the internal operation of MDM, **all indexes start at 0**. This means that the domain of this state factor is $\mathcal{X}_1 = \{0, \ldots, 5\}$.

```
robot_mls.addStateFactor (StateDep()
                          .add ("PersonIsWaiting"));
```

This binds the second state factor to the predicate `''PersonIsWaiting"`, which means that $x_2 = 1$ iff the predicate is true.

```
spin();
```

This spins this node's thread in a loop, during which the State Layer will be running and listening for predicate updates. Currently, the state description cannot be changed after the `spin()` function is called. If the state description contained in a State Layer does not match what is expected by an associated Control Layer, a warning will be thrown.

3.2 The Observation Layer

In the partially observable case, sensorial information should be mapped to a space of discrete factored *observations*, by an *Observation Layer*. The resulting decision making loop, which can be used for POMDP-driven agents, is shown in Figure 3.

The most significant difference of this abstraction layer with respect to a fully-observable State Layer is that, while in the latter case states are mapped directly from assignments of persistent predicate values; in an Observation

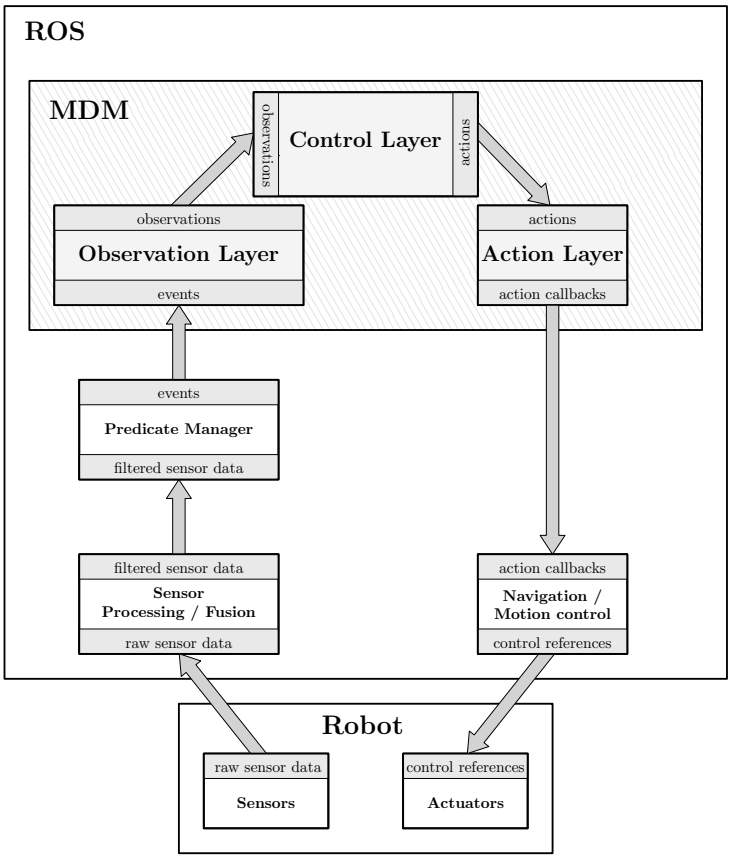

Figure 3: The basic control loop for a POMDP-based agent using MDM.

Layer, observations are mapped from instantaneous *events*. The rationale is that an *observation* symbolizes a relevant occurrence in the system, typically associated with an underlying (possibly hidden) state transition. This event is typically captured by the system as a change in some conditional statement. The semantics of events, therefore, are defined over instantaneous conditional changes, as opposed to persistent conditional values.

The Predicate Manager package also makes it possible to define named events. These can be defined either as propositional formulas over existing predicates, in which case the respective event is triggered whenever that formula becomes true; or directly as conditions over other sources of data (*e.g.* a touch sensor signal).

Observation spaces can also be factored. Observation factors are always associated with at least two events, and so their definition is similar to that of integer-valued state factors. Let $\mathcal{E} = \{e_1, \ldots, e_n\}$ be a set of events and $\mathcal{O} = \{\mathcal{O}_1, \ldots, \mathcal{O}_m\}$ a factored observation space description. An Observation

Layer defines a mapping $\mathcal{G}_i : \mathrm{PS}(\mathcal{E}) \rightarrow \mathcal{O}_i$, for each observation factor i, that is, it maps from subsets of $\mathcal{E}$ directly onto each discrete domain $\mathcal{O}_i$. Let $\nu_t(\mathcal{E}) \subseteq \mathcal{E} \cup \emptyset$ represent the events in $\mathcal{E}$ which have triggered at time t (possibly none). Then, an observation factor i is defined through a set $\mathcal{E}' = \{e'_1, \ldots, e'_k\} \subseteq \mathcal{E}$ of ordered *asynchronous events* – that is, at most one event in $\mathcal{E}'$ triggers at any given instant ($\max_{t \in \mathbb{R}_0^+} |\nu_t(\mathcal{E}')| = 1$) – so that, iff $\nu_t(\mathcal{E}') \neq \emptyset$, $o_i|_t = \kappa_t$ with $\kappa_t = \min\{k : e'_k \in \nu_t(\mathcal{E}')\}$. Less formally, whenever an event is received, the value of the i-th observation factor is the index of the first (and supposedly only) active event in the associated event set.

Note that, although the definition of *event* which is here used implies a natural asynchrony (events occur randomly in time), this does not mean that synchronous decision making under partially observability cannot be implemented through MDM. As it will be seen, only the Control Layer is responsible for defining the execution strategy (synchronous or asynchronous) of a given implementation.

As in the case of the State Layer, an Observation Layer always outputs the *joint* value of the observation factors that it contains.

3.2.1 Implementing an Observation Layer

An example of the implementation of a simple Observation Layer in ROS will now be analyzed. In this example, there is a single observation factor, defined over events which are associated to a robot's task of patrolling its environment for fires.

```
#include <markov_decision_making/ObservationLayer.h>

using namespace ros;
using namespace markov_decision_making;

int main (int argc, char** argv)
{
  init (argc, argv, "observation_layer_example");

  ObservationLayer ol;
  ol.addObservationFactor (ObservationDep()
                           .add ("Elevator Hallway Clear")
                           .add ("West Corridor Clear")
                           .add ("Coffee Room Clear")
                           .add ("South Corridor Clear")
                           .add ("LRM Clear")
                           .add ("Soccer Field Clear")
                           .add ("Found Fire"));

  spin();

  return 0;
}
```

Breaking down the code:

```
ObservationLayer ol;
```

This declares an Observation Layer which silently subscribes to event topics coming from Predicate Manager (`~/event_updates` and `~/event_map`). The Observation Layer isn't started until the `spin()` function is called. Before that is done, however, the observation space description must be provided.

```
ol.addObservationFactor (ObservationDep()
                         .add ("Elevator Hallway Clear")
                         .add ("West Corridor Clear")
                         .add ("Coffee Room Clear")
                         .add ("South Corridor Clear")
                         .add ("LRM Clear")
                         .add ("Soccer Field Clear")
                         .add ("Found Fire"));
```

This creates our observation factor and associates it to a set of named events (which will be flowing in from `~/event_updates`). As before, indexes start at 0 – the output of this Observation Layer is 0 when the event `"Elevator Hallway Clear"` is received.

While spinning, the resulting observation is published asynchronously to the `~/observation` topic, whenever one of the associated events is caught. The Observation Layer also listens in the `~/observation_metadata` topic for incoming observation space descriptions from associated Control Layers, for validation purposes.

3.3 The Control Layer

The Control Layer is responsible for parsing the policy of an agent or team of agents, and providing the appropriate response to an input state or observation, in the form of a symbolic action.

The internal operation of the Control Layer depends on the decision theoretic framework which is selected by the user as a model for a particular system. For each of these frameworks, the Control Layer functionality is implemented by a suitably defined (multi)agent *controller*. Currently, MDM provides ready-to-use controllers for MDPs and POMDPs operating according to deterministic policies, which are assumed to be computed outside of ROS/MDM (using, for example, MADP).

For POMDPs, the Control Layer also performs belief updates internally, if desired. Consequently, in such a case, the stochastic model of the problem (its transition and observation probabilities) must typically also be passed as an input to the controller (an exception is discussed in Section 4). The system model is also used to validate the number of states/actions/observations defined in the System Abstraction and Action Layers.

MDM Control Layers use the MADP Toolbox extensively to support their functionality. Their input files (policies, and the model specification if needed), can be defined in all MADP-compatible file types, and all decision-theoretic frameworks which are supported by MADP can potentially be implemented as MDM Control Layers. The MADP documentation describes the supported file types for the description of MDP-based models, and the corresponding policy representation for each respective framework.

The Control Layer of an agent also defines its real-time execution scheme. This can be either synchronous or asynchronous – synchronous execution forces an agent to take actions at a fixed, pre-defined rate; in asynchronous execution, actions are selected immediately after an input state or observation is received by the controller.

All agent controllers can be remotely started or stopped at run-time through ROS service calls. This allows the execution of an MDM ensemble to be itselft abstracted as an "action", which in turn can be used to establish hierarchical dependencies between decision-theoretic processes (see Section 3.4).

3.3.1 Implementing a Control Layer

The following examples show how generic Control Layers for MDPs and POMDPs can be implemented. First, for an asynchronous (event-driven) MDP:

```
#include <string.h>

#include <ros/ros.h>

#include <markov_decision_making/ControllerEventMDP.h>

using namespace std;
using namespace ros;
using namespace markov_decision_making;

int main (int argc, char** argv)
{
  init (argc, argv, "mdp_control_layer_example");

  if (argc < 4) {
    ROS_ERROR_STREAM ("Usage: mdp_control_layer_example"
                      << "<number of states>"
                      << "<number of actions>"
                      << "<path to MDP Q-table>");
    abort();
  }

  size_t nr_states = atoi(argv[1]);
  size_t nr_actions = atoi(argv[2]);
  string q_table_file = argv[3];

  ControllerEventMDP controller (nr_states,
                                 nr_actions,
                                 q_table_file);

  spin();

  return 0;
}
```

The `ControllerEventMDP` class implements an asynchronous controller for an MDP agent. Note that it requires, as an input, the $Q-$value function associated with the desired policy. The MDP stochastic models are not required, only its domain sizes (number of states and actions). However, if the MDP itself is defined in a MADP-compatible file format, it can be passed as an input instead (see the MDM API for alternate constructors). In that case, the model will be parsed by MADP, and the following options can be

set as parameters in the node's private namespace:

- `is_sparse` (boolean): when set to `true`, the internal representation of the transition and observation functions of the model uses sparse (`boost::uBLAS` matrices). Typically, for large models, this results in faster parsing and less memory usage at run-time.

- `cache_flat_models` (boolean): when set to `true`, even if the model is defined in a factored format (for example, if the model is written in the *ProbModelXML* format), the "flat" (non-factored) version of the transition and observation functions will be calculated and stored in memory.

MDP controllers will subscribe to the `~/state` topic and publish the associated action to the `~/action` topic. Additionally, if the MDP model is provided, the controller will publish the immediate reward after an action selection to the `~/reward` topic.

In contrast, the following implementation describes an asynchronous POMDP controller:

```
#include <string.h>

#include <ros/ros.h>

#include <markov_decision_making/ControllerEventPOMDP.h>



using namespace std;
using namespace ros;
using namespace markov_decision_making;



int main (int argc, char** argv)
{
  init (argc, argv, "pomdp_control_layer_example");

  if (argc < 3) {
    ROS_ERROR_STREAM ("Usage: pomdp_control_layer_example"
                      << "<path to problem file>"
                      << "<path to POMDP value function>");
    abort();
  }

  string problem_file = argv[1];
  string value_function_file = argv[2];
```

```
  ControllerEventPOMDP controller (problem_file,
                                   value_function_file);

  spin();

  return 0;
}
```

Note that, for POMDP controllers operating according to the scheme shown in Figure 3, the problem file must be passed to the constructor, so that it is able to handle belief updates at run-time. POMDP controllers receive observations through the `~/observation` topic. Additionally, they subscribe to `/~initial_state_distribution`, which can be used to set the initial belief of the POMDP. As outputs, POMDP controllers publish actions on the `/~action`; the belief state at run-time to `/~current_belief`; and the immediate (expected) reward to `~/reward`.

3.4 The Action Layer

Symbolic actions can be associated with task-specific functions through an MDM Action Layer. In this layer, the user univocally associates each action of a given (PO)MDP with a target function (an action *callback*). That callback is triggered whenever a Control Layer publishes its respective action. An Action Layer can also interpret commands from a Control Layer as *joint actions*, and execute them under the scope of a particular agent.

The general purpose of an action callback is to delegate the control of the robotic agent to other ROS modules outside of MDM, operating at a lower level of abstraction. For example, action callbacks can be used to send goal poses to ROS native navigation modules; or to execute scripted sets of commands for human-robot interaction. However, the Action Layer also makes it possible to abstract other MDM ensembles as actions. This feature allows users to model arbitrarily complex hierarchical dependencies between MDPs/POMDPs (see Figure 4 for an example of the resulting node layout and respective dependencies).

The layered organization of MDM, and the "peer-to-peer" networked paradigm of ROS, allow action *execution* and action *selection* to be decoupled across different systems in a network, if that is desired. This can be accomplished by running an agent's Action Layer on a different network location than its respective Control Layer. For mobile robots, this means that the components of their decision making which can be computationally expensive (sensor processing / abstraction, and stochastic model parsing, for

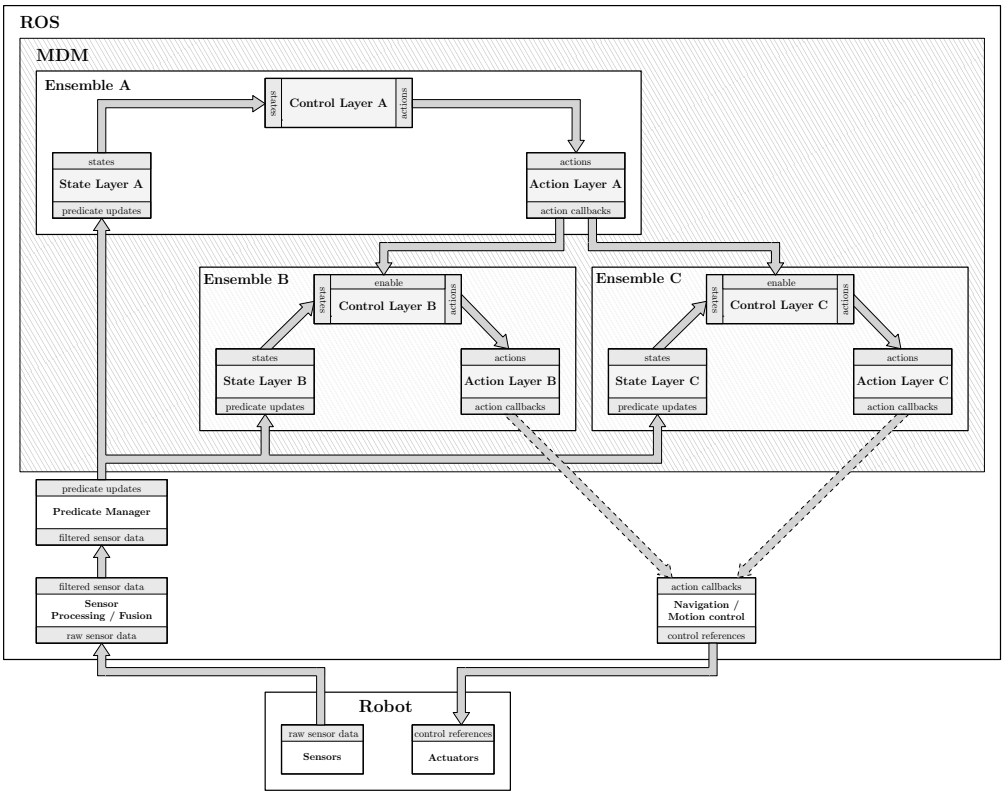

Figure 4: An example of the organization of a hierachical MDP, as seen by MDM. The actions from Action Layer A enable / disable the controllers in ensembles B and C. When a controller is disabled, its respective State Layer still updates its information, but the controller does not relay any actions to its Action Layer.

example), can be run off-board. For teams of robots, this also implies that a single Control Layer, implementing a centralized multiagent MDP/POMDP policy, can be readily connected to multiple Action Layers, one for each agent in the system (see Figure 5).

For the implementation of typical topological navigation actions (which are prevalent in applications of decision theory to robotics), MDM includes *Topological Tools*, an auxiliary ROS package. This package allows the user to easily define states and observations over a map of the robot's environment, such as those obtained through ROS mapping utilities. It also allows the abstraction of that map as a labeled graph, which in turn makes it possible to easily implement context-dependent navigation actions (*e.g.* "Move Up", "Move Down") in an MDM Action Layer.

3.4.1 Implementing an Action Layer

The following example shows how a simple MDM Action Layer can be implemented. The present Action Layer interprets the high-level actions *Patrol*, *Assistance Response*, *Trespassing Response* and *Emergency Response*. The former of these is itself a POMDP (this is an example of hierarchical control), while the remaining actions are carried out by finite-state controllers defined using the SMACH[1] package.

```
#include <boost/bind.hpp>

#include <ros/ros.h>
#include <std_srvs/Empty.h>

#include <markov_decision_making/ActionLayer.h>

using namespace ros;
using namespace markov_decision_making;

class Actions
{
public:
  Actions () :
    patrol_stop_client_
    (nh_.serviceClient<std_srvs::Empty>("patrol_POMDP/stop")),
    patrol_reset_client_
    (nh_.serviceClient<std_srvs::Empty>("patrol_POMDP/reset")),
    assistance_SMACH_client_
    (nh_.serviceClient<std_srvs::Empty>("assistance_SMACH/act")),
    trespassing_SMACH_client_
    (nh_.serviceClient<std_srvs::Empty>("trespassing_SMACH/act")),
    emergency_SMACH_client_
    (nh_.serviceClient<std_srvs::Empty>("emergency_SMACH/act")){}

  void patrolPOMDP() {
    std_srvs::Empty e;
    patrol_reset_client_.call (e);
  }

  void assistanceSMACH() {
    std_srvs::Empty e;
    patrol_stop_client_.call (e);
    assistance_SMACH_client_.call (e);
  }

  void trespassingSMACH() {
    std_srvs::Empty e;
```

[1] http://www.ros.org/wiki/smach

```
    patrol_stop_client_.call (e);
    trespassing_SMACH_client_.call (e);
  }

  void emergencySMACH() {
    std_srvs::Empty e;
    patrol_stop_client_.call (e);
    emergency_SMACH_client_.call (e);
  }
private:
  NodeHandle nh_;
  ServiceClient patrol_stop_client_;
  ServiceClient patrol_reset_client_;
  ServiceClient assistance_SMACH_client_;
  ServiceClient trespassing_SMACH_client_;
  ServiceClient emergency_SMACH_client_;
};

int main (int argc, char** argv)
{
  init (argc, argv, "action_layer_example");

  Actions am;
  ActionLayer al;
  //Patrol Action
  al.addAction (boost::bind (&Actions::patrolPOMDP, &am));
  //Assistance Response Action
  al.addAction (boost::bind (&Actions::assistanceSMACH, &am));
  //Trespassing Response Action
  al.addAction (boost::bind (&Actions::trespassingSMACH, &am));
  //Emergency Response Action
  al.addAction (boost::bind (&Actions::emergencySMACH, &am));

  spin();

  return 0;
}
```

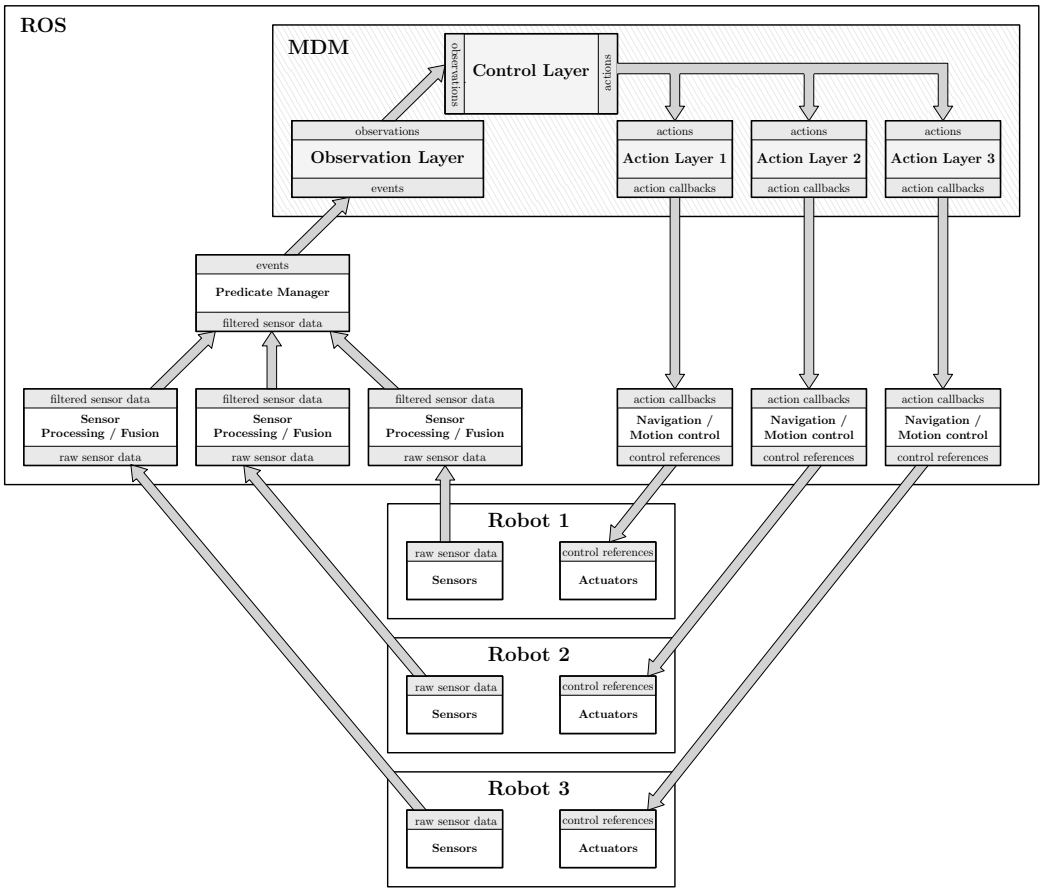

Figure 5: An MPOMDP implemented by a single MDM ensemble – in this case, there are multiple Action Layers. Each one interprets the joint actions originating from the centralized controller under the scope of a single agent. There is a single Observation Layer which provides joint observations, and a single Predicate Manager instantiation which fuses sensor data from all agents.

The most important aspect of this example is how actions are bound to functions in the action layer:

```
Actions am;
ActionLayer al;
//Patrol Action
al.addAction (boost::bind (&Actions::patrolPOMDP, &am));
//Assistance Response Action
al.addAction (boost::bind (&Actions::assistanceSMACH, &am));
//Trespassing Response Action
al.addAction (boost::bind (&Actions::trespassingSMACH, &am));
//Emergency Response Action
al.addAction (boost::bind (&Actions::emergencySMACH, &am));
```

This will create an Action Layer with four associated actions, where each of them is implemented by a method in the auxiliary `Actions` class. Note that the latter class is not a part of MDM – it is simply a design choice, a generic way of containing all action implementations in the same object. The `addAction(.)` function accepts `boost::function` pointers (to functions with no arguments / return values), so using `bind` on the methods of our auxiliary class directly returns the desired type. Those methods will be immediately called when their respective action is received through the `~/action` topic. For example, receiving action 0 will trigger the `Actions::patrolPOMDP()` method.

In this example, all action-bound functions hand over the control of the agent to other modules through ROS service calls. This is also a design choice – using services lets the Action Layer know when and if its client modules receive a request for execution, so it allows for a more secure control of the program. The execution on the client side is outside of the scope of the Action Layer (and of this example). Each SMACH finite-state controller is triggered by calling an `<>\act` service in its respective namespace, which can advertised by a SMACH *Service State*[2]. The execution of a lower-level MDP/POMDP can also be controlled via service requests: Control Layers automatically advertise services to *stop*,*start* or *reset* their execution (the latter essentially stops and starts the controller from its initial conditions). In this particular implementation, the lower-level POMDP (`patrol_POMDP`) is controlled by calling its `stop`/`reset` services.

The downside to the service-based approach is that a client which should *not* be running in a given context may also need an explicit request to stop its execution. This is true, in particular, for the lower-level MDPs/POMDPs.

[2]see: http://www.ros.org/wiki/smach/Tutorials/ServiceState

4 Deploying MDM: Considerations for Specialized Scenarios

The previous Sections covered the internal organization of MDM and provided an overview of its implemantation in generic scenarios. The present Section discusses how MDM can be applied to scenarios with practical requirements which lie outside of the more common deployment schemes which have been presented so far.

4.1 POMDPs with External Belief States

In some scenarios, the probability distribution over the space state of the system can be handled indirectly, and continuously, by processes which can run independently of the agent's decision-making loop. This may be the case, for example, if a robot is running a passive self-localization algorithm, which outputs a measure of uncertainty over the robot's estimated pose, and if the user wants to use that estimate, at run-time, as a basis for the belief state of an associated POMDP.

In MDM, this execution scheme can be implemented by omitting the Observation Layer of a POMDP agent, and instead passing the estimated belief state directly to its Control Layer. For this purpose, POMDP Control Layers subscribe, by default, to the `~/external_belief_estimate` topic. Upon receiving a belief estimate, the Control Layer can output its associated action, according to the provided policy file, either asynchronously or at a fixed temporal rate. A representation of this deployment scheme is shown in Figure 6.

There are, however, notable caveats to this approach:

- The description of the state space used by the Control Layer must be known by the belief state estimator. In particular, the notion of *state* that is assumed by the module which responsible for the belief estimation must be the same as that which is modeled in the POMDP. If this is not the case, then the probability distributions originating from the external estimator must first be projected onto the state space of the POMDP, which is not trivial. The system abstraction must be carried out within the belief state estimator;

- Algorithms which produce the belief state estimate directly from sensor data (*e.g.* self-localization) typically operate synchronously, at the same rate as that source data. This means that asynchronous, *event-driven* POMDP Control Layers are not well-defined in this case;

- Planning is still assumed to be carried out prior to execution, and, during planning, the stochastic models of the POMDP are assumed to be an accurate representation of the system dynamics. If the external belief updates do not match the sequences of belief states predicted by the transition and observation models of the POMDP, then the agent can behave unexpectedly at run-time, since its policy was obtained using ill-defined models; if, on the other hand, the external belief updates are consistent with those which are predicted by the POMDP model, then a better option (in terms of the amount of work required for the deployment of the POMDP) is to simply carry out those belief updates internally in the Control Layer, as per the deployment scheme described originally in Section 3.2.

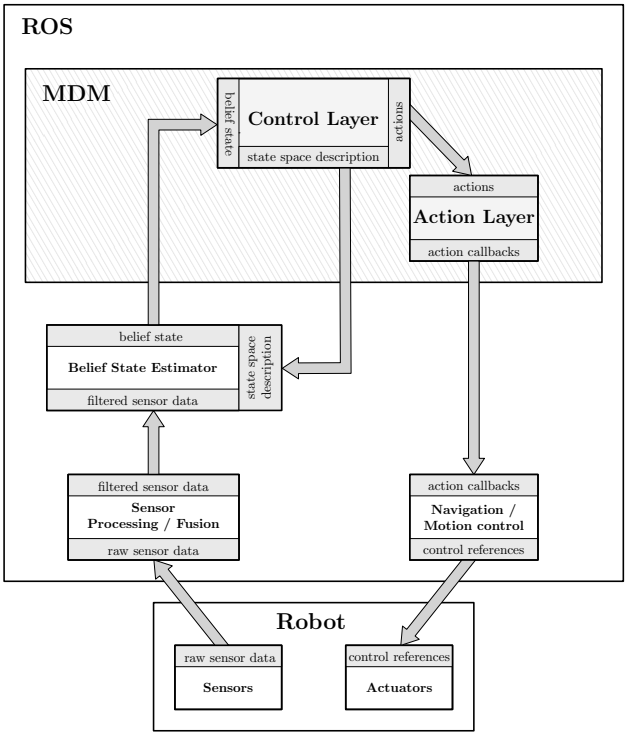

Figure 6: A deployment scheme for a POMDP-based agent where belief updates are carried out outside of MDM. The node that is responsible for the estimation of the belief state must have the same state space description as the Control Layer. The MDM ensemble is driven at the same rate as the belief state estimator, so it is implicitly synchronous with sensorial data.

4.2 Multiagent Decision Making with Managed Communication

For multiagent systems, the standard operation of ROS assumes that a single ROS *Master*[3] mediates the connection between the all of the nodes that are distributed across a network. After two nodes are connected, communication between them is peer-to-peer, and managed transparently by ROS, without providing the user the possibility of using custom communication protocols (for example, to limit the amount of network traffic). Therefore, in its default deployment scheme, the ROS Master behaves as a single point of failure for the distributed system, and more advanced communication protocols such as the one proposed in [1] cannot be used.

A more robust multiagent deployment scheme combines multiple ROS Masters, with managed communication between them (see Figure 7). Typically, each mobile robot in a multi-robot team will operate its own ROS Master. Topics which are meant to be shared between Masters through managed communication must be explicitly selected and configured by the user.

For MDM centralized multiagent controllers, which completely abstract inter-agent communication, using a multimaster deployment scheme simply means that a complete copy of its respective MDM ensemble must be running locally to each Master. Communication should be managed before the System Abstraction Layer, ideally by sharing Predicate Manager topics, so that each agent can maintain a coherent view of the system state, or of the team's observations. Note that, since each agent has local access to the joint policy, it can execute its own actions when given the joint state or observation.

For MDM controllers with explicit communication policies, communication should be managed at the output of the System Abstraction Layer, possibly with feedback from the Control Layer. The rationale in such a case is that states or observations are local to each agent (as opposed to being implicitly shared), and may not contain enough information for an agent to univocally determine its own action at the Control Layer. Consequently, the Control Layer should also be capable of fusing system information arriving from diferent sources.

[3] http://www.ros.org/wiki/Master

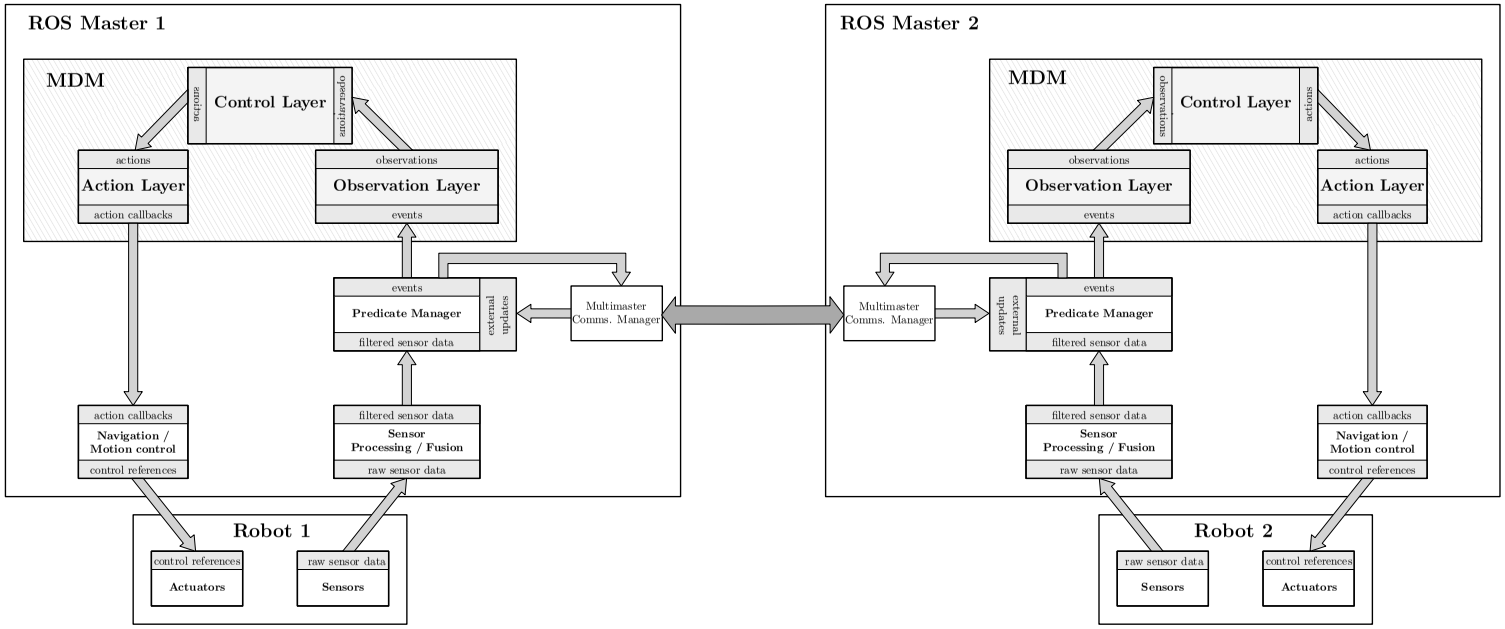

Figure 7: A multiagent MDM deployment scheme with multiple ROS Masters, and managed multimaster communication. Predicate Manager topics are explicitly shared between ROS Masters, so each Observation Layer still outputs *joint* observations. This is the approach which was taken in the MAIS+S project, using SocRob Multicast [1] to manage the multimaster communication.

5 Software Documentation and Maintenance

With the MDM package installed, the `rosdoc` tool will generate a local copy of its documentation. The MDM `C++` API can then be consulted, and its respective documentation includes further detail on the practical implementation of each MDM module.

MDM is maintained by João Messias (jmessias@isr.ist.utl.pt) and the Predicate Manager package is maintained by João Reis (joaocgreis@gmail.com). Please contact the authors if you wish to contribute to these packages, or if you require further help in their usage.

References

[1] J. C. G. Reis. Distributed communications system for multi-robot systems. Master's thesis, Instituto Superior Técnico, 2012.