EnMAS: A New Tool for Multi-Agent Systems Research and Education

Connor Doyle and Martin Allen
Computer Science Department
University of Wisconsin-La Crosse
La Crosse, WI 54601
connor.p.d@gmail.com; mallen@cs.uwlax.edu
http://enmas.org/

Abstract

Multi-agent systems (MAS) is a field of growing relevance to our modern world with applications everywhere from heterogenous distributed computing and storage to robot swarms. One model of MAS is the Decentralized, Partially Observable Markov Decision Process, or Dec-POMDP, used extensively in current research, and capable of modeling both large multi-agent and smaller single-agent systems, whether stochastic or deterministic.

The project to be demonstrated is a framework and an application for specifying Dec-POMDP problems and agents. Named EnMAS (Environment for Multi-Agent Simulation), the project is designed to be useful for research and as a teaching tool. To that end, Dec-POMDP problems are specified using Scala, an expressive hybrid functional and object-oriented language that targets the Java Virtual Machine. Agents may be written in Java or Scala. While the latter makes code-writing more efficient, backwards compatibility is provided for those who want to write AI agents in Java, a feature designed to make the framework more useful in the classroom setting.

Algorithms for doing on-line learning have different input requirements than those for off-line planning, and satisfying both can lead to error-prone code duplication. This problem is addressed by the ability to employ a single EnMAS problem specification syntax in both phases. Another major goal of the EnMAS project is to unify the formats used to encode experiments. Efforts are made toward a clean, human-readable problem specification syntax, and code for both problem and agent are archived using the common JAR file format for easy sharing. An additional goal is to provide high performance scalability. EnMAS is a client-server application, where each client in turn may host many agents. In this way, users may run both server and clients on a single node and still reap the benefits of today's multi-core architectures. Alternatively, more machines can run as an ad-hoc cluster.

This work is the product of a Master's project in software engineering. Sample results to do with Dec-POMDP research come from an undergraduate research project in the area.

1 Introduction

Multi-agent systems (MAS) is a subject of much ongoing research in artificial intelligence (AI). In such problems, agents are either software or hardware entities, working in some shared environment. Such problems are then differentiated based on such things as (i) the number of agents involved, (ii) whether the state dynamics are deterministic or stochastic, (iii) whether the environment is fully or partially observable, or (iv) whether agents are working together cooperatively, or operating in competition. In cooperative settings, a popular general problem framework is the *decentralized, partially observable Markov decision process* (Dec-POMDP); when competitive, such problems can be modeled by the slightly more general *partially observable stochastic game* (POSG).

For AI researchers and educators, the challenge of working in such environments is twofold. First and foremost, there is the difficulty posed by the computational complexity of the problems themselves, and the challenge in finding tractable algorithms for the generation of effective agent policies. A secondary challenge arises from the associated complexity of support software needed to do research and to teach students about the area. Leaving aside the code needed to specify solution algorithms, much research time must go into the generation of problem domains, the application of algorithms to those domains, and the simulation of algorithm runs over the cases required to establish empirical results. Similarly, for the AI instructor, considerable time can be needed to create simulation and visualization tools for given problems, so students can then learn how to implement various agents and their associated solution methods. In both arenas, code is generally written for individual uses, without standardization, hampering the ability to share benchmark problems, agent solution algorithms, and simulation/visualization products.

The EnMAS (Environment for Multi-Agent Simulation) software system provides an open-source set of tools for both researchers and educators. Users provide the encoding of a problem environment, using a straightforward, standardized format. Agents can then be written in a choice of languages, interacting with that environment through a basic software interface. Once this is completed, the server-based system handles all simulation activity, generating outcomes in response to agent action choices, and returning necessary observations and reward values back to those agents. A clean graphical user interface provides access to simulation functionality, without direct interaction with the underlying code-base.

Standardized code formats and interfaces make any problem domain or agent implementation easily sharable using common JAR code bundles, providing "plug and play" ability to swap in new problems and solution technologies, and promoting common research benchmarks. The EnMAS framework eliminates much time-consuming effort in generating support code. For educational purposes, the creation of code for agents (typically the role of students) can be separated neatly and completely from that for the environment and support (typically the role of instructors). At a deeper level, the system allows the user to reduce costly and error-prone code duplication between the generation of the problem domain and the creation of solution agents. Overall, the system speeds the development of MAS problem simulations, and makes the process less prone to errors.

2 Formal Problem Specification

Users of the EnMAS system will typically begin by formulating an agent-based AI problem in the form of a decentralized partially observable Markov decision process (Dec-POMDP). These models are highly general, allowing the user to represent any number of problems to be solved using techniques ranging from full forward contingency planning to experience-based reinforcement learning algorithms.

Definition 1 (Dec-POMDP)**.** A *decentralized partially observable Markov decision process*, $\mathcal{D}$, is specified by a tuple:

$$M = \langle \{\alpha_i\}, S, \{A_i\}, P, \{\Omega_i\}, O, R, T \rangle$$

with individual components as follows:

- Each α_i is an *agent*; S is a finite set of *world states* with a distinguished *initial state* s^0; A_i is a finite set of *actions*, a_i, available to α_i; Ω_i is a finite set of *observations*, o_i, for α_i; and T is the (finite or infinite) *time-horizon* of the problem.
- P is the *Markovian state-action transition function*. $P(s, a_1, \ldots, a_n, s')$ is the probability of going from state s to state s', given joint action $\langle a_1, \ldots, a_n \rangle$.
- O is the *joint observation function* for the set of agents, given each state-action transition. $O(a_1, \ldots, a_n, s', o_1, \ldots, o_n)$ is the probability of observing $\langle o_1, \ldots, o_n \rangle$, if joint action $\langle a_1, \ldots, a_n \rangle$ causes a transition to global state s'.
- R is the *global reward function*. $R(s, a_1, \ldots, a_n)$ is the reward obtained for performing joint action $\langle a_1, \ldots, a_n \rangle$ when in global state s.

Dec-POMDPs generalize the single-agent POMDP model, which, along with its fully observable cousin, the MDP, has been used extensively as a foundation for AI work, as described by Dean et al. [6] and by Kaelbling, Littman, and Cassandra [10]. As mentioned, the model is highly general. By varying model parameters, a variety of different AI problem domains can be generated.

- If the number of agents is restricted to one, then the problem becomes a POMDP.
- If the combination of agent observations uniquely determines the underlying state-space, then the problem is a Dec-MDP; if each agent actually observes the global state, then the problem is a multiagent MDP, as described by Boutilier [3]. If the state is observed directly in a single-agent problem, then it is an MDP.
- Deterministic domains can easily be modeled by adjusting the state transition function to match.
- Non-cooperative problems can be modeled by supplying separate reward functions for distinct agent types, rather than a single joint reward as given; in the more general context, the problems are then partially observable stochastic games (POSGs), as described by Hansen, Bernstein, and Zilberstein [9].

While not the only model for MAS research, Dec-POMDPs are increasingly popular, and are in many cases equivalent to other possibilities; for more information about the relationships between some of these models, see the paper of Seuken and Zilberstein [17].

A potential solution to an AI problem formulated as a Dec-POMDP is a policy, dictating what actions each agent should take based upon its observations.

Definition 2 (Policies)**.** A *local policy* for an agent α_i is a mapping from sequences of that agent's observations, $\overline{o_i} = \langle o_i^1, \ldots, o_i^k \rangle$, to its actions, $\pi_i : \Omega_i^\star \rightarrow A_i$. A *joint policy for* n *agents* is a collection of local policies, one per agent, $\pi = \langle \pi_1, \ldots, \pi_n \rangle$.

A solution method for a decentralized problem seeks to find some joint policy that maximizes expected value given the starting state (or distribution over states) of the problem. While dynamic programming algorithms that generate optimal policies for Dec-POMDPs exist, these methods are limited in their applicability due to the fundamental complexity of the problem domain, shown by Bernstein, et al. [2] to be complete for the class NEXP (nondeterministic exponential time). Recent research focuses on finding approximate solution techniques, such as those explored by Seuken, Carlin, and Zilberstein [4, 15, 16].

2.1 An Example Dec-POMDP

To make the model more concrete, we give a simple example of a problem domain and its representation in Dec-POMDP form. Variants of the *box-pushing problem*, as shown in Figure 1, have been used to test a variety of algorithmic techniques for decentralized problem solving. In this problem domain, two agents need to collaborate in a simple grid-world to push boxes into a goal region. While smaller boxes can be pushed by a single agent, larger boxes require that both agents push together. A variant of the problem can be formalized as follows:

- The agent set consists of two agents α_1 and α_2; each agent has the same set of actions, allowing it to *move* in one of four cardinal directions, or to *wait*.
- Each state $s \in S$ consists of a sequence of variables recording the (x, y) locations of each agent, and of three boxes, two small and one large. The space is discrete, consisting of a fixed number of grid locations.
- The action transitions given by P combine deterministic and stochastic outcomes. When agents *move* into an unoccupied region, their actions always succeed; when they attempt to *move* out of the field of play, or into a square containing another agent, or simply *wait*, they always remain where they are. When pushing boxes (by moving into squares occupied by the box), however, things are stochastic, with the move action succeeding with probability 0.8 and failing with probability 0.2; in addition, the large box can only be moved if both agents work together to push it in the same direction at once, and this action succeeds with a joint probability of 0.64, failing with probability 0.36. When boxes are pushed into the *Goal* region along the top of the space, the box disappears, and a new one re-spawns at the original location near the bottom of the screen.

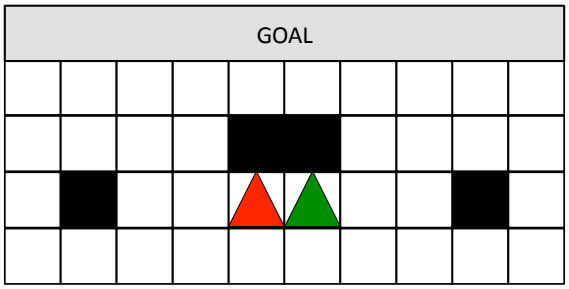

Figure 1: A 2-agent box-pushing problem. The agents, shown as triangles, must work together in order to push the large black box into the goal location. The smaller boxes can be pushed by a single agent working alone. At this point in time, the agents have succeeded in moving the large box one location in the direction of the goal area.

- The domain is partially observable, as each agent only observes what is contained in each of the four squares immediately adjacent to its own (*empty*, *box*, *agent*, *wall*). Even if one has access to the observations of both agents, then, certain parts of the state-space remain unobserved, since each agent can only observe the location of at most one of the three boxes, for instance.
- The joint reward r at each time-step is a sum total of rewards for each agent. Any successful *move* or *wait* action produces a penalty of -1, whereas any unsuccessful *move* (due to collision with a wall or other agent, or a failed box-push) generates a penalty of -5. Pushing a small box into the goal area produces a reward of $+50$, while pushing the large box into the goal produces a joint reward of $+250$.

Given the dynamics of the system, an optimal policy for the problem is to continuously push large boxes into the goal region, taking the shortest collision-free route back from the top of the screen to the bottom each time (in order to minimize the movement penalties). While this simple optimal policy can not be generated by a guaranteed-optimal algorithm, given the high computational complexity of the problem domain, and the considerable combinatorics even in such a simple example, approximating methods can often arrive at that best policy, and recent research of our own (forthcoming) has suggested that multi-agent reinforcement learning can work well in such domains.

3 The EnMAS System

EnMAS is a concurrent, distributed, open source Dec-POMDP engine. It also provides an API for defining POMDP problem domains, agents, and other ”iteration subscribers” such as loggers and graphical output components. The main goal influencing the system design is to enable rapid prototyping of executable POMDP problem domains for research and teaching purposes, while abstaining from making assumptions about the problem domain. Equally important is to maintain consistency between the implementation and formal theoretical models. The result is a truly generic Dec-POMDP framework.

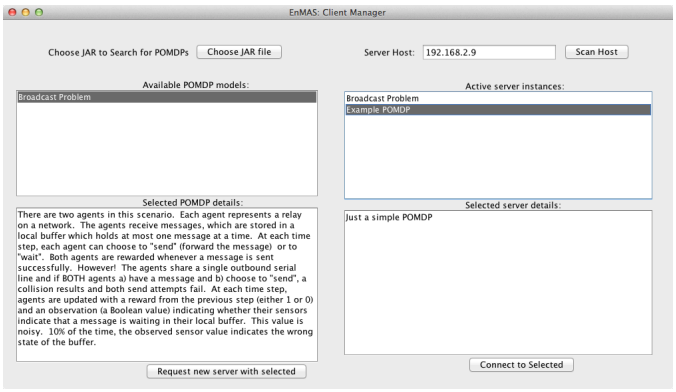

Figure 2: The EnMAS *Client Manager*. Using this GUI interface, the user chooses one or more problem instance specifications. Each such problem is then loaded by the chosen server. Connections are requested to the server instances so that agents can begin to interact with the problem in simulation. This example shows the manager connected to a server running instances of two example domains supplied with EnMAS, a very simple testing instance, and a version of the well-known Dec-POMDP benchmark broadcast problem [14].

3.1 How EnMAS works

EnMAS consists of two main components: client and server. Neither can run without user-supplied code. The server needs code that describes a POMDP, and the client needs code that describes the behavior of AI agents as well as how to log and visualize each iteration of a running experiment. This user code is supplied to the system in JAR format. Code samples for agents and POMDP problems are provided with the distribution that users may learn to use the system without making a significant time investment. These samples also highlight the modular nature of the system.

The EnMAS server is capable of creating a POMDP instance from a coded specification sent to it over the network. A server is launched on its host machine, and then the user interfaces with that server via the *Client Manager* interface, choosing problem instances to run and connecting with those running instances, as shown in Figure 2. Once the POMDP instance is created, clients may connect to it via the *Session Manager* interface, as shown in Figure 3. This program, launched automatically when the user connects to the server, allows the user to choose agent types to run. Once the number of agents required by the problem domain have been loaded properly, the system begins to iterate its simulation. At each step of the iteration, the system waits for each agent to submit a chosen action—without the server needing to be aware in any way how those actions are chosen—and then generates the state transition, observations, and rewards, all according to the problem specification. This information is returned from the server, and the manager programs ensure that each agent receives its own observations and rewards, again as given by the problem specification.

The Session Manager also features the ability to attach *iteration subscribers* to a server problem instance. These subscribers receive the ongoing data from the problem and can be written by the user to do things like logging relevant information or displaying graphical

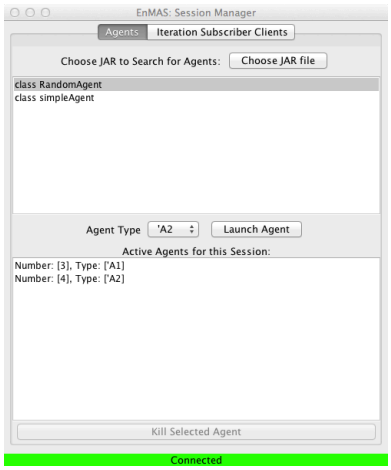

Figure 3: The EnMAS *Session Manager*. This interface, launched when the Client Manager connects with a server running a problem instance, allows the user to choose agents to run in simulation on the problem. This example shows a pair of agents, of two distinct types. Also shown is the tab from which the user can select Iteration Subscribers, which allow for logging or graphical representation of the outputs of the simulation process.

output. Currently, EnMAS comes with a number of simple agents for its sample problem types, along with elementary loggers for doing things like tracking agent reward as they interact with the problem.

3.2 The EnMAS Problem Specification Format

In large part, the EnMAS problem format hews closely to the formal definition of a Dec-POMDP already given, with a few small exceptions to allow ease in adding multiple types of agents, and full generality to the competitive case. One notion supported by the EnMAS POMDP specification API that is not present in the traditional definition is the *agent type*, by which it is possible to partition the agents into groups, each of which share a common reward function and action set. This allows for problems in which all agents are of the same type, which matches the typical Dec-POMDP, where there is joint reward and agents typically have the same available actions in most states; at the same time, we can handle problems like POSGs, where agents compete, and possess distinct reward functions and capabilities. Depending on the nature of the problem domain, it is also possible to take the agent type into account when writing the observation function. This allows for easier coding when many distinct agents are involved, since the user only has to write code to deal with each type of agent, rather than explicitly handling each individual agent separately.

In order to define a problem, the EnMAS user writes a subclass of the supplied code, `org.enmas.pomdp.POMDP`, using the Scala language. To do so, the user provides:

- A name for the problem and a problem description.
- A list of minimum and maximum cardinalities for each agent type. (The server will

	Formal Dec-POMDP	EnMAS
Transition Function	$S \times \langle a_1 \dots a_n \rangle \times S \to \mathbb{R}$	$(State, JointAction)$ $\to Either[State, List[(State, Int)]]$
Reward Function	$S \times \langle a_1 \dots a_n \rangle \times S \to \mathbb{R}$	$(State, JointAction, State)$ $\to (AgentType) \to Float$
Observation Function	$S \times A \times S \times \Omega_n \to \mathbb{R}$	$(State, JointAction, State)$ $\to (Int, AgentType) \to Observation$

Table 1: Comparison of type signatures of elements from formal Dec-POMDP model and the EnMAS Dec-POMDP Specification Format.

iterate when and only when the set of active agents chosen satisfies the minimum numbers for each type.)

- The initial state from which the problem begins.
- The actions function, which takes an agent type and returns a set of actions.
- The transition function, which takes the current state and a joint set of actions, and returns either a single $State$ object, or a list of $(State, Int)$ tuples, where the integer values are to be interpreted as a normalized probability distribution.
- The observation function, which takes the previous state, the joint action set, and the current state as arguments. The observation function returns another function that takes an agent ID (an integer) and an agent type as arguments and returns an $Observation$. (Here $Observation$ is simply a type alias of $State$.)
- The reward function, which takes the previous state, the joint action set, and the resulting state as arguments. The reward function also returns a function, which takes an agent type as its single argument and returns a floating point value.

Table 1 compares the formal Dec-POMDP definition with the matching functions in EnMAS. One aspect that requires particular explanation is the variable return type of the transition function, which can either return an individual state, or a set of states with accompanying distribution. The reason for this flexibility is that in some situations (mostly forward planning algorithms) it is necessary to inspect the probability distribution underlying the Dec-POMDP explicitly. Often, this can lead the researcher to write two separate pieces of code to do the same basic work. On the one hand, it is often most convenient to code the transition function used in the simulation so that, given a state and joint action, it returns the next state according to the prescribed transition probabilities given by the model. When creating a planning agent, on the other hand, it can be necessary that the agent know the full set of possible outcome states, and their underlying distribution, and so a second piece of code is often written to generate this more detailed information.

This leads to two potential problems. First, there is the duplicate effort to write two pieces of code where one might do the job just as well. Second, there is the possibility for error, since the codes will have fundamentally different structures, and so the user may inadver-

tently introduce divergences between the transition model assumed by the agent and the one that is actually used in the simulation. In EnMAS, to eliminate these potential pitfalls, the user can, if they choose, write the transition model for the simulation in the more explicit form, and the system itself will then return back the actual state-to-state transitions based upon the encoded distribution. In such cases, the user can simply use the same function in the problem specification for simulation as they do in the agent specification for planning, only needing to write (and get right) the explicit distribution code once. At the same time, when not using algorithms that require full access to the distribution (such as when doing experience-based reinforcement learning, say), the requirement to write the code for state transitions in this explicit form can be very onerous, especially where the state-set is large and the dynamics complex. Thus, EnMAS allows the user to write the problem specification of the transition function in the potentially simpler form that returns but a single state at each iteration. This flexibility should allow researchers and educators to tailor the form of their coding to their specific needs, significantly simplifying things where possible, while providing better quality assurance where required.

3.3 EnMAS: The Agent Specification Format

In order to define an AI agent for use in a simulation, users write a subclass of the supplied `org.enmas.client.Agent`. All else being equal, it is preferable that the agent is written in Scala, like the problem environment. However, support is provided to allow the subclass to be written in Java, making it easier for less experienced programmers to write agents, something that can be particularly handy in the classroom context, where the student might be responsible only for the agent code, to be run on an environment supplied by the instructor. Things are also made easier by keeping the elements necessary for the agent specification relatively simple. Agents come with convenience methods to get their own number and agent type as well as the set of available actions. The user-written agent definition is then responsible for providing:

- A name for the agent.
- A policy function, which takes as arguments an Observation and a Reward and returns an Action.

At each iteration of the framework, each agent's policy function is called. Users can therefore make this function as simple or as complex as needed, so long as it takes the proper inputs and returns some possible Action as output at every call. The Action returned by each agent is forwarded back to the server, which then computes the next iteration.

3.4 How does EnMAS achieve the goals set out in the introduction?

As already described, EnMAS is meant to provide a convenient tool for research and teaching about multi-agent decision problems. We highlight here some of the features that we believe make it especially useful for those purposes. In particular, we examine ways in

which the model used in EnMAS is closely tied to the formal Dec-POMDP framework, while also allowing some flexibility in coding. We also examine how the distinction of the process for coding agents from that for problem environments, provides advantages for code sharing and simplifies the process of writing agents.

3.4.1 Consistency with the Formal Model

The EnMAS problem format sticks relatively close to the formal definition of a Dec-POMDP. For instance, the form of the reward function matches well with that given by the definition, with the exception that distinct classes of agents may share rewards, which allows more generality to competitive or team-based problem domains.

In the formal definition of a Dec-POMDP, the signature of the transition and observation functions are mathematically tidy. However, due to the fact that the EnMAS system needs to actually simulate problem runs iteratively, it reformulates these signatures to make them easier to specify and compute. In the case of the observation function, stochasticity is encapsulated within the function definition. The Int in the signature of the returned function is an agent number, corresponding to one agent active within the model.

The transition function is somewhat more complicated. For some types of solution approaches involving planning, agents need to inspect the transition function in order to compute a policy. However, for other problem domains the state space may grow very large. In those cases it can be difficult to express the state transition probabilities explicitly. Thus, as already described in Section 3.2, the user is given two possible function signatures from which to choose, allowing them to write the transition code in the form that suits them best. In user-supplied transition functions that return a list of $(State, Int)$ tuples, the probability of each State component is given in normalized form, i.e., the corresponding Int component divided by the sum of all given Int components. States with non-positive Int components are ignored as impossible transitions. In the event that the transition function returns the empty list, the next state is equal to the current.

The Scala programming language was helpful when implementing this flexibility in the API. The Scala standard library provides a generic disjoint union type, $Either[+A, +B]$. $Either$ has exactly two concrete subclasses: $Left[+A, +B]$ and $Right[+A, +B]$. The EnMAS server applies the transition function to its arguments and uses Scala's language-level pattern matching facility to determine the type of the result. Two implicit conversion functions with signatures $State \rightarrow Either[State, List[(State, Int)]]$ and $List[(State, Int)] \rightarrow Either[State, List[(State, Int)]]$ make the entire process transparent to API users.

3.4.2 No Assumptions: The EnMAS State Class

In the typical Dec-POMDP definition, the state set S is simply some finite set of atomic units, without any explicit internal structure. This is fine for many purposes, but there are instances where factoring the state into component variables is more useful. EnMAS supports both approaches in the same way. In all cases, the EnMAS State class is a generic,

fully type-safe, immutable string-keyed hash map. Again, the Scala programming language proved useful during implementation. The three most important methods of the State class are $+$, $-$, and $getAs$.

The $+$ method takes a $(String, T)$ as input, where T is the implicit type parameter. To work around JVM type erasure, $+$ takes an additional implicit parameter, of type $Manifest[T]$. $Manifest$ is part of the Scala reflection API. The $+$ method returns a $State$ object containing a binding from the given string to a wrapper containing the object reference and the $Manifest$ for that object's type. It thus serves to add a variable-value pair to an existing state object, defining the state in terms of the mappings it contains. For "atomic" states, each state might need only a single variable, taking it to some form of distinguishing ID value, allowing us to distinguish states s_i, $s_j \in S$ for the purposes of writing the problem specification functions. (The $-$ method is similar, removing mappings from some state.)

The signature for $getAs$ is as follows: $getAs[T](key : String)(implicit\, m : Manifest[T]) : Option[T]$. The type parameter is required. The $Option[+T]$ monad is Scala's answer to the Java convention of returning nullable object references. It has exactly two concrete subclasses: $Some[+T]$ and $None$. $getAs$ returns a $Some$ if there exists a binding from the supplied key to an object of type T, and $None$ otherwise. The calling code may pattern match the result, a powerful feature common to functional languages like Scala, which can be used to write many of the functions over states needed in the problem specification.

3.4.3 Simplified and Separated Agent Specifications

EnMAS is designed to operate in a modular fashion, with code for problem specifications and agent implementations loaded separately by the server and client sides, respectively. This structure provides a number of nice features. In particular, it makes code sharing simpler, since a particular benchmark problem is now distinct from any particular algorithmic technique used by solution agents, and users can thus easily swap out one problem for another using the same basic agent technology, or do the opposite and evaluate a number of distinct solution techniques over a single benchmark.

This separation also makes writing agents a much simpler thing, since users only have to pay attention to the observations the agent can receive and the actions it can choose, without needing to even look at the code for the environment itself. In addition, to further expand the usefulness of EnMAS as an educational tool, the API supports writing agent specifications in Java as well as Scala. By design, most Scala methods are callable from Java code, unless they rely on language features that Java lacks. One more complex method is the $State.getAs$, described above. In such cases, efforts are made to conform to existing Java conventions. For example, the Java version of $getAs$ takes a prototype object instead of an explicit type parameter. Instead of returning an $Option$ object (which would be unwieldy in the calling code without pattern matching), the method simply returns an object conforming to the static type of the prototype object. In cases where the Scala version would return $None$, the Java version throws a $NoSuchElementException$, which can be handled (or not) as desired by the user.

4 Related Work

While other simulation environments and frameworks for MAS exist, EnMAS is distinct in that it is particularly targeted to AI work using Markov decision processes and their multi-agent generalizations. By keeping the functional structure of code as close as possible to the formal definition of the problem class, EnMAS is intended to make it easier for a researcher or instructor to set up problem domains, specify agents, run simulations, and share results and techniques. This makes EnMAS distinct from a project like *breve* [11, 12], for instance, which supplies very strong tools for coding agents and for providing sophisticated 3D graphics, but uses the idea of an *agent* as a primitive. In this approach, all interacting elements of a simulation environment are defined as agents, governed by computed physics. While very impressive results are possible, the lack of separation between agent and environment can make it difficult to represent common AI decision problems in a natural way. EnMAS is also distinct from a more ambitious simulation/visualization environment like *MASON* [8, 13], which is very well suited for large-scale simulations of very simple agents, as in swarm computation or cellular automaton approaches, but can require sophisticated programming talents to set up the overall simulation. While EnMAS currently does not come with the impressive pre-packaged graphics capabilities of either of these projects, we believe that it stands as a useful tool for researchers and educators in AI especially.

Previous efforts have been made to standardize formats for encoding benchmark POMDP and Dec-POMDP problems [1, 5]. Such formats have primarily focussed on simplified, matrix-based tabular syntax for such things as the state-transition probabilities. As a result, it can be sometimes difficult to specify problems of higher complexity, due to the sheer size of the tables involved. In EnMAS the use of a genuine programming language like Scala or Java, along with standardized interfaces to the problem and agent types, has both its pros and cons. On the one hand, writing code for EnMAS is just that, and so there is the inevitable learning curve and associated complexity. On the other hand, the use of a functional language like Scala can make the specification of things like transition or observation functions more natural, and often much more compact, allowing for more complex problem dynamics. In addition, these prior efforts have simply supplied problem specifications, without any interest in a supporting code framework for performing simulations, and so any benchmark problem must still be re-coded in a language of choice for incorporation into an research or teaching project. In EnMAS, on the other hand, the system allows us to do away with time-consuming and error-prone recoding of existing problems or agents. Since the system is pre-built to handle all simulations based on passed-in code products, any benchmarks, once written, can simply be shared as-is, and loaded directly into one's own local install of EnMAS, without any need for re-writing whatsoever.

5 Future Work

The current EnMAS distribution comes with a small set of sample problems, consisting of simple sample Dec-POMDP instances (including one simple but popular benchmark, the

distributed broadcast channel problem [14]). Similarly, each problem has been supplied with some relatively simple agent types, employing fixed policies of action (stochastic or deterministic, but not in any deep sense intelligent or adaptive). These initial examples are useful for initial demonstration purposes, and can help new users manage the initial process of setting up and testing their EnMAS install. They also serve as code samples for those who wish to begin writing their own problems and agents. To increase the usefulness of the tool, and to extend its appeal to the research and education community, we are making ongoing efforts to add to this code-base. Our current intention is to supply a wider range of benchmark problems, along with some in-system implementations of some established algorithms. Code packages will be hosted and distributed on-line, with room for new users to contribute their own examples and ideas. We also intend to expand the existing set of iteration subscriber code instances, providing examples and resources for researchers who want to do in-simulation data processing and educators looking to add graphical visualization in order to make AI ideas more transparent to students.

Ultimately, the goal is to provide a common platform for the community interested in decision-theoretic planning and learning under uncertainty, easing the spread of benchmark problems, allowing direct comparison of new and existing algorithms, and making collaborative research much easier to sustain across different groups and institutions.

6 EnMAS Development and Acknowledgments

This project was created and is maintained by Connor Doyle [7] in partial fulfilment of the Master of Software Engineering degree at the University of Wisconsin-La Crosse under the advisement of Drs. Martin Allen and Kenny Hunt.

The authors acknowledge the generosity and support of the Department of Computer Science at the University of Wisconsin-La Crosse and the National Science Foundation, without which this project would not be possible.

References

[1] Christopher Amato. Dec-POMDP website. `http://rbr.cs.umass.edu/~camato/decpomdp/`. Resource Bounded Reasoning Lab, Computer Science Department, University of Massachusetts at Amherst.

[2] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27(4):819–840, 2002.

[3] Craig Boutilier. Sequential optimality and coordination in multiagent systems. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 478–485, Stockholm, Sweden, 1999.

[4] Alan Carlin and Shlomo Zilberstein. Value-based observation compression for DEC-POMDPs. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems*, pages 501–508, Estoril, Portugal, 2008.

[5] Anthony Cassandra. POMDP website. `http://www.cs.brown.edu/research/ai/pomdp/`. Computer Science Department, Brown University.

[6] Thomas Dean, Leslie Pack Kaelbling, Jak Kirman, and Ann Nicholson. Planning under time constraints in stochastic domains. *Artificial Intelligence*, 76:35–74, 1995.

[7] Connor Doyle. EnMAS project website. `http://enmas.org/`.

[8] Evolutionary Computation Laboratory/Center for Social Complexity. MASON website. `http://cs.gmu.edu/~eclab/projects/mason/`. George Mason University.

[9] Eric A. Hansen, Daniel S. Bernstein, and Shlomo Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence*, pages 709–715, San Jose, California, 2004.

[10] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101:99–134, 1998.

[11] Jon Klein. Breve website. `http://www.spiderland.org/`.

[12] Jon Klein. BREVE: a 3D environment for the simulation of decentralized systems and artificial life. In *8th International Conference on the Simulation and Synthesis of Living Systems*, pages 329–335, Sydney, NSW, Australia, 2002.

[13] Sean Luke, Gabriel Catalin Balan, Liviu Panait, Claudio Cioffi-Revilla, and Sean Paus. MASON: A multiagent simulation library. In *Agent 2003 Conference on Challenges in Social Simulation*, pages 59–64, Chicago, IL, 2003.

[14] James M. Ooi and Gregory W. Wornell. Decentralized control of a multiple access broadcast channel: Performance bounds. In *Proceedings of the Thirty-Fifth Conference on Decision and Control*, pages 293–298, Kobe, Japan, 1996.

[15] Sven Seuken and Shlomo Zilberstein. Improved memory-bounded dynamic programming for decentralized POMDPs. In *Proceedings of the Twenty-Third Conference on Uncertainty in Artificial Intelligence*, Vancouver, British Columbia, Canada, 2007.

[16] Sven Seuken and Shlomo Zilberstein. Memory-bounded dynamic programming for DEC-POMDPs. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 2009–2015, Hyderabad, India, 2007.

[17] Sven Seuken and Shlomo Zilberstein. Formal models and algorithms for decentralized decision making under uncertainty. *Autonomous Agents and Multi-Agent Systems*, 17(2):190–250, 2008.