The Past and Future of Multiagent Systems

José M Vidal*
Computer Science and Engineering
University of South Carolina
Columbia, SC, 29208
vidal@sc.edu

Paul Buhler
Computer Science
College of Charleston
Charleston, SC, 29424
pbuhler@cs.cofc.edu

Hrishikesh Goradia
Computer Science and Engineering
University of South Carolina
Columbia, SC, 29208
goradia@cse.sc.edu

Abstract

We describe the lessons learned from using various technologies as aids in teaching a graduate multiagent systems class. The class has been offered six times over the last five years. The technologies described are RoboCup (along with our Biter and SoccerBeans tools), NetLogo, JADE, and FIPA-OS. We also discuss our view of the future of multiagent systems which includes the separation of software agent design into a separate class that focuses on distributed programming and the development of a unifying notation for representing multiagent problems.

1. Introduction

Over the last decade the field of multiagent systems has evolved from its beginnings in distributed artificial intelligence as a discipline largely concerned with distributed problem solving, to the mature discipline we see today which brings together researchers from disciplines as diverse as economics, game theory, biology, robotics, software engineering, and artificial intelligence. All this diversity make is difficult to provide students with the background needed for understanding multiagent approaches to problem-solving. This difficulty has been further compounded by the recent creation of sub-fields of study such as auction-based systems, robotic-based systems, software agents, and mechanism design approaches. Nonetheless, we believe that it is possible to provide students with a deep understanding of the foundations of multiagent theory and algorithms by using a coherent notation and providing them with hands-on experience.

This article describes the lessons we have learned over five years of teaching multiagent systems. Section 2 describes our use of RoboCup, NetLogo, FIPA, and formal frameworks along with the lessons we have learned from using them. Section 3 describes our future plans which include the continuing use of RoboCup and NetLogo, the separation of software agents into its own class, and the development of a unifying notation for multiagent systems.

2. Past Classes

The first author has taught a graduate class in multiagent systems at the University of South Carolina six times during the years 1999–2003. On average, the class is attended by 10–20 students each semester. Since it is graduate class in the Computer Science and Engineering department, the students are assumed to be competent programmers and to have some mathematical sophistication. There are no formal prerequisites for the class, specifically, an Artificial Intelligence class is not a prerequisite. The class has used the Weiss [23] and the Wooldridge [25] textbooks, supplemented with other readings.

Since its inception, the class has echoed research in the field by having an explicit separation between the part that deals with theory and algorithms and the part that deals with software and hardware agents. One research community emphasizes mathematics and logic to produce models, theories, and algorithms. The other research community emphasizes software engineering and ontologies to produce

* This material is based upon work supported by the National Science Foundation under Grant No. 0092593.

software or hardware systems. This split mirrors the traditional split we see in computer science between the theory and algorithms researchers and the software engineers. Of course, both sides depend on each other. The system engineers need the algorithms developed by the theoreticians and the theoreticians need the problems raised by the system engineers. Both parts present their own particular challenges for a teacher of multiagent systems.

The theory and algorithms of multiagent systems differs from that of traditional computer science in its significant dynamic component. That is, the algorithms can often be properly understood only by visualizing the agents engaging in their individual actions and then recognizing the emergent behavior that results. In fact, many of the algorithms lack steady-state solutions and only provide an expected dynamic behavior.

The difficulty in building agent systems can only be properly understood by actually building fairly complex systems. The development process helps the students understand the difficulty in predicting the emergent behavior that will result from simple agent behaviors as well as the difficulty in debugging a distributed asynchronous system.

As such, our class uses a hands-on approach to teaching multiagent systems. We strive to get the students to build systems so they can see the algorithms in action and thus achieve an intuitive understanding of how local changes affect the emergent behavior of the system. In order to meet this aim we have sought out and used various tools that allow students to gain first-hand experience within the time constraints of a one semester class. We have also tried to merge the use of these tools with the theory presented in the textbooks. The next few sections describe our experiences with the various tools we have used.

2.1. RoboCup, Biter, and SoccerBeans

The class has used the RoboCup simulator since the second time the class was taught. The use of RoboCup as a teaching tool has been very successful and it is something we have already written about [21, 20]. The students are made to form teams of one to three students. The teams compete in a RoboCup simulated soccer tournament at the end of the semester. The students are given a month to complete their assignments. The grade for the project is determined by the team's standing in the tournament and the quality of their writeup. In the tournament all teams play all other teams so as to minimize the likelihood of an unlucky loss at an early game and to gather more information on the teams' techniques. The tournament format has proven to be a great motivator. The familiarity of the problem domain allows the students to immediately start working on their problem solving techniques rather than spending time trying to understand the problem. Their techniques

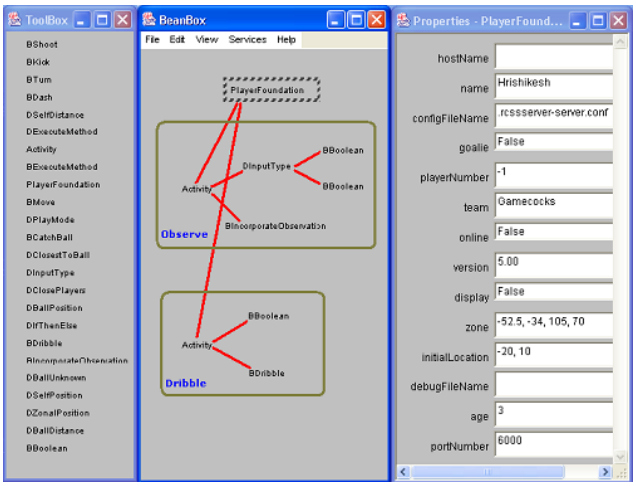

Figure 1: Screenshot of the SoccerBeans system.

usually echo those used by humans, but are always modified to the particular limitations of the two dimensional and discrete soccer simulator.

After the first class we learned that the basic RoboCup client offers so little functionality that students had to spend almost all their time trying to implement basic behaviors such as dribbling and passing the ball instead of focusing on the multiagent aspects of the problem. In order to ease this burden we developed the Biter system [6]. Biter implements a basic player that maintains a world model where all the objects are given absolute coordinates and also implements many useful behaviors such as kicking, passing, dribbling, and catching a pass. These features were chosen because all students agreed they were essential for a working player.

In the most recent class we used the SoccerBeans [10] development system, as shown in Figure 1. The SoccerBeans system takes basic Biter player behaviors along with decision criteria such as the player's absolute position, the ball's absolute position, the player's distance from the ball, the number of teammates or opponents closing in, etc., and wraps all of them as Java Beans. The students can then use Sun's Bean Development Kit to build their agents by visually connecting the various beans in different ways. This system allows the students to fully concentrate on conducting their experiments and requires little or no coding on their part.

We successfully used the SoccerBeans software in the last class. The students found it very easy to use. In fact, most of the final teams were developed without the need to write a single line of code. However, the develop-test cycle proved to be very slow. It took them around 5–10 minutes to compile a new agent because all the Beans had to be compiled every time. This problem is an artifact of the Soc-

```
to setup
  ca
  create-n-turtles num-turtles
end

to move
  locals [cx cy]
  set cx mean values-from turtles [xcor]
  set cy mean values-from turtles [ycor]
  set heading towardsxy cx cy
  if (distancexy cx cy < radius) [
    set heading heading + 180]
  if (abs distancexy cx cy - radius > 1)[
    fd speed / 1.414]
  set heading towardsxy cx cy
  ifelse (clockwise) [
    set heading heading - 90]
  [
    set heading heading + 90]
  fd speed / 1.414
end

to update
  no-display
  while [count turtles > num-turtles][
    ask random-one-of turtles [die]]
  ask turtles [move]
  display
end
```

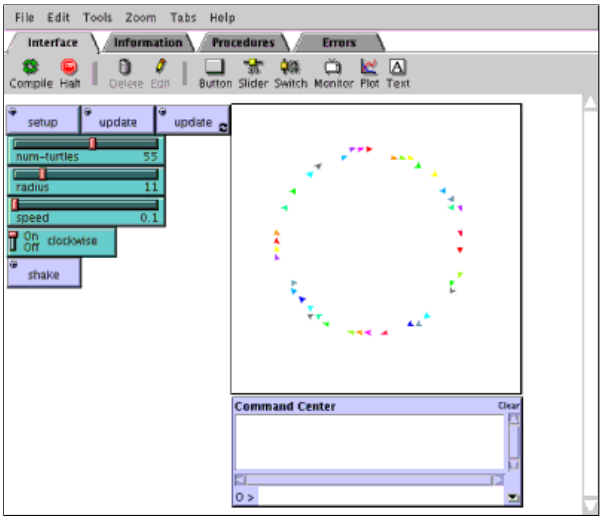

Figure 2: Screenshot of NetLogo running the Circle program in which all turtles self-organize to run around in a circle, along with *all* the NetLogo code needed to implement this application.

cerBeans architecture and will either have to be fixed for the next class or we will have to revert back to Biter. We also note that the winning team only used the Bean Development Kit for their first version of the agent which they then refined by directly editing the code. In this manner, they eliminated the long compile time and were able to perform more tests, which resulted in a better team.

2.1.1. Lessons Learned Our experience using RoboCup as a learning tool has been overwhelmingly positive. Its use along with Biter or SoccerBeans allows students to directly experience the problems inherent in building multiagent systems. Many students have commented on their surprise when, by making a small change to one of the players they inadvertently destroyed an emergent behavior that seemed completely unrelated. We have been witness to the long hours they devote to implementing techniques they thought would help them win the game. Their final write-ups have also confirmed their ability to think in terms of a system behavior emerging out of simpler behaviors.

There are, however, some problems with the use of RoboCup. The techniques developed for this domain are unlikely to transfer to other domains, even other robotic domains. Very few of the standard multiagent algorithms are applicable to this domain. Most notably, since the domain features cooperative agents it does not utilize research into selfish agents. These hindrances stem from the fact that RoboCup is a very well defined problem while multiagent research provides solutions and techniques for many different types of problems.

We are also not entirely satisfied with our Biter and SoccerBeans implementations. In order for RoboCup to be a multiagent problem the players need to be able to make long-distance passes. These passes require good kicking and catching behaviors, as well as good dribbling behaviors so that the player can set itself correctly for making a pass. Biter implements these behaviors reasonably well, but there is a some room for improvement. Specifically, we would like to have a dribbling behavior that allows the player to *consistently* keep the ball within its kickable area, a passing behavior that *consistently* places the ball at the requested coordinates, and an intercept behavior that goes to where the ball will be and stops it. We believe that with these behaviors available the students will have to focus much more on the team dynamics rather than making marginal improvements to the basic behaviors. We have been unable to work on these improvements due to lack of resources.

2.2. NetLogo

In order to provide our students with experiences in many other problem domains, the last two classes have incorporated the use of NetLogo [24]. NetLogo is a descendant of StarLogo, which is a parallel version of Logo, which is a variant of Lisp designed to teach children the basics of programming. StarLogo was designed by Resnick to be a programming language for teaching children the distributed mindset [15]. Resnick's hypothesis is that children have an inherent tendency to explain the world using a centralized mindset—when asked how ants deliver food back to the nest they invariably answered that the ants simply followed orders from the queen—which can be corrected by letting them play with simulations of decentralized phenomena. Once the children were able to program simulated ants following a pheromone trail they understood how the emergent behavior arises from simple interactions.

NetLogo is based on StarLogo and is written in Java. NetLogo's designers also have a didactic mission but focus instead on older students and social scientists as their core user groups. The NetLogo interface allows the user to easily plot different types of bar and line graphs, to keep track of the value of any variable, and to easily change the value of variables. Figure 2 shows a screenshot of a simple application along with all the code needed to implement it. The NetLogo language has grown to be very sophisticated. The NetLogo mailing list is actively populated by social scientists who use it to build agent-based models of interactions. Cognizant of its use as a research tool, the NetLogo programmers have been careful to use good randomization and floating point libraries so that experiments result in the same outcome regardless of the platform in which they are run.

The NetLogo language and metaphor are powerful enough to easily represent many of the traditional problems in multiagent systems. We have a webpage[1] which contains NetLogo implementations of many of these problems as well as a few more recent algorithms. Included in this page are implementations of the Adopt [11] algorithm for graph coloring and N-queens problems, asynchronous backtracking [26] for the N-queens problem, the mailmen problem [16], tileworld [14], asynchronous weak commitment [23] for the N-queens problem, path-finding using pheromones [13], a distributed recommender system [18], reciprocity in package delivery [17], the coordination game [8], and congregating [4]. These implementations provide ample evidence of NetLogo's expressive power.

We used NetLogo both as a way to demonstrate the functioning of some algorithms during the lecture, and as the platform for the implementation of various small assignments. We had one lecture, usually the second day of classes, which introduced the NetLogo metaphor and the basics of the language. The students easily understood the turtle and patches metaphor and quickly became experts in the language. A few students had difficulty comprehending NetLogo's use of lists and mapping functions over lists; this was their first exposure to a functional language (they only knew Java or C++). However, by the end of the first assignment all students were fluent in NetLogo.

The students were given five or six NetLogo assignments each semester. They were given two weeks to either implement a solution to a well-known problem or to modify a well-known solution so as to achieve a particular improvement. Some of the topics covered were: implement a solution to the tileworld problem, solve an instance of the distributed sensor problem, solve the mailman problem, modify an existing solution to a package delivery problem in order to develop and incentive compatible protocol, and implement modifications on asynchronous backtracking and weak-commitment search for solving the distributed N-queens problem. Some of the assignments asked the students to provide solutions to well-known open problems in multiagent research. These allowed the students to apply the techniques learned in class and think about the tradeoffs inherent to any solution. Other assignments asked them to implement variations on well-known algorithms. These gave them deeper insight into the workings of the algorithms.

2.2.1. Lessons Learned The NetLogo assignments gave the students the ability to see many of the traditional multiagent algorithms in action. The students could modify various parameters and observe how these changes were reflected in the emergent behavior. The develop-test cycle was thus reduced to mere seconds and new ideas could be implemented in minutes. The students' response was largely positive. However, a few complained that the assignments required too much time—an unfortunate side effect of the open-ended nature of the assignments. We found that we had to make it very clear how much work was expected for each assignment otherwise some students would spend endless hours perfecting their solution.

We consider our NetLogo experiment a success. It is the only suitable platform we have found that combines a tiny learning curve, incredible expressive power, and an extremely short develop-test cycle. The students also found the ability to quickly draw a GUI and plot graphs a great help in extending and debugging their programs. NetLogo brings to life many of the multiagent problems and algorithms taught in class. It also serves as a nice complement to RoboCup. While RoboCup stresses the real-world requirements of uncertainty and near real-time responses, NetLogo allows us to focus on the algorithm itself by providing a world with no uncertainty or real-time requirements.

1 `http://jmvidal.cse.sc.edu/netlogomas/`

Of course, there are a few drawbacks to NetLogo. While the turtle and patches metaphor is very flexible, it cannot represent some multiagent systems such as those involving agents on the Internet. This shortcoming is not exclusive to NetLogo but is simply a symptom of our inability to visualize these type of multiagent systems. There is no commonly agreed upon standard visual language for representing software agents and their interactions, even when we limit their interaction to, say, buying and selling. If such a standard language is developed then it should not be too hard to provide a NetLogo implementation.

Another problem we encountered was our inability to describe the problem domain in the NetLogo language itself, that is, to provide a data abstraction layer. NetLogo does not provide any object-oriented data encapsulation tools. As such, the students always have direct access to the underlying metaphor and can get confused about which actions are legal in a particular assignment. For example, in the sensor domain we told the students that their agents could only see the area close to them, but we had no way to express this constraint programmatically in such a way that the students could not program an agent that could see things far away, neither could we programmatically check to make sure that this constraint had not been violated. This lack of encapsulation made grading difficult.

Finally, we also missed the availability of a Java API for adding our own computationally intensive subroutines. Specifically, we are interested in extending the agents with machine learning subroutines and subroutines for finding equilibriums in game matrices but the computational requirements of these subroutines would make them very slow if implemented in NetLogo. Fortunately, the authors have announced such an API is forthcoming.

2.3. FIPA Agents

Our class always covers the FIPA architecture. We describe the overall architecture and discuss several interaction protocols. In the Fall 2001 and Fall 2002 classes we also had assignments that involved using either JADE [1] or FIPA-OS [5]. In these assignments, groups of two or three students developed a small distributed meeting scheduling application using their chosen FIPA-compliant agent framework. Each agent represents a user of the system. The students had to develop and implement exchange protocols that enabled the agents to schedule meetings that maximized the users' utilities. The users had utility values over the times of day they wanted the meeting scheduled. Also, in all meetings there were some users whose attendance is required and others whose attendance was optional.

2.3.1. Lessons Learned We found that the students preferred to use JADE for their implementations. They thought that JADE's documentation was better and its API allowed them to quickly get something that works. However, both systems had significant learning curves which prevented the students from dedicating more time to development of the interaction protocols. Almost all of the final systems met the minimum requirements as set by the assignments but were otherwise uninspired. The interaction protocols did not take into account many of the possible ways in which people can lie in order to get what they want. The systems neglected to consider the possibilities of some agents going offline for significant periods of times or other network failures.

Because the class has a RoboCup tournament at the end, it was hard to find the time for another long programming project. As such, the FIPA assignment was dropped in the Fall 2003 class. We also feel uncertain about the future of the FIPA protocols given the popularity of web services. We felt that we were wasting the students' time by forcing them to learn a complex API that was soon to be obsolete. Instead, the most recent class only taught the theoretical basics of the FIPA architecture—ideas that will probably be re-implemented as web service technologies. We explain our reasoning further in Section 3.1.

2.4. Theory

The first three classes used the Weiss textbook [23] and the last two used the Wooldridge textbook [25]. Both books cover roughly the same theoretical material. They start by describing the same formal agent model and then proceed to cover the topics of agent architectures, game theory, auctions, coordination, voting, and learning in multiagent systems. Weiss does a better job at covering distributed constraint satisfaction algorithms, so we have continued to use that chapter. We also found an unpublished textbook by Vlassis [22] which contains excellent introductions to game theory and mechanism design. Both of these chapters were also used in the latest class.

2.4.1. Lessons Learned Since multiagent theory brings together research from economics, game theory, formal logic, and theoretical AI, it lacks a comprehensive notation. The three textbooks mentioned try to develop a common notation for all the algorithms and protocols they present but largely fail in this effort. Specifically, both Weiss and Wooldridge present a notation for describing an agent—its inputs, actions, and environment—but fail to present a consistent notation for describing the desired system behavior. Vlassis comes close to achieving a common notation but it still not entirely satisfactory. As such, our strategy has been to teach all the new notation required for each topic. This strategy has the advantage that the students become fluent in the language of various disciplines but it has the disadvantage that some students fail to see how the various multiagent techniques relate to each other.

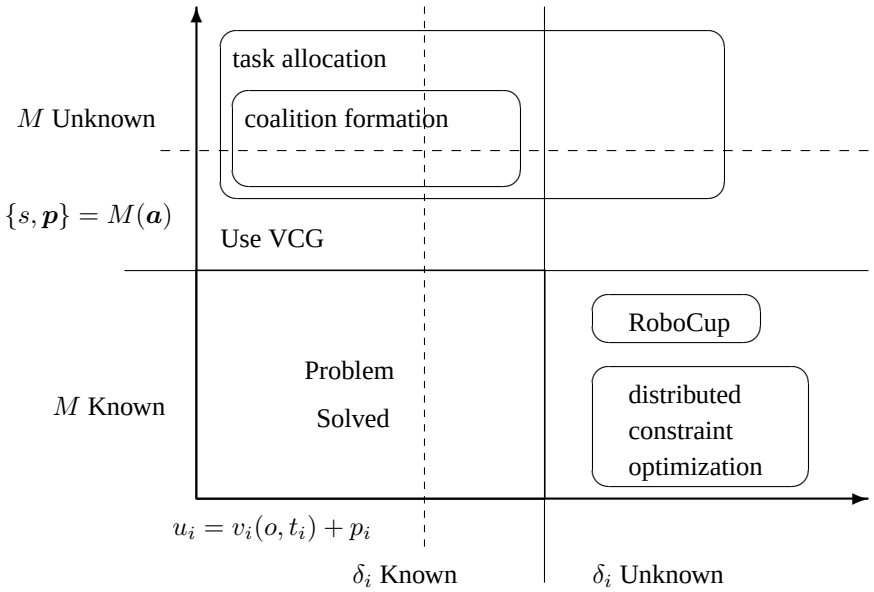

Figure 3: Space of all multiagent problems. The x-axis represents whether the decision function δ_i of each agent is known or unknown to the designer. The y-axis represents whether the mechanism function $M(\boldsymbol{a}) \rightarrow o$ is known or unknown to the designer.

3. The Future

We can spot two trends that will deeply impact multiagent systems: the growing popularity of the semantic web along with web services and the growing popularity of mechanism design. The first trend affects how multiagent systems are used in the real world, while the second trend affects how we, as researchers, talk about the theory and algorithms behind them. We also see the second trend as a harbinger of a unifying formal framework for multiagent systems.

3.1. The Semantic Web

Since the elucidation of the semantic web vision [2], the web services community and parts of the multiagent community have grown closer together. We believe that the work done by the multiagent researchers on software agents, interoperability, ontologies, and architectures (FIPA) will be absorbed by the web services community, as exemplified in the W3C "Web Services Architecture" working group note [3]. We know that our students are largely interested in SOAP and associated technologies because of market demands. We also know that these technologies are complex. As such, the multiagent system class can no longer explain these technologies in depth. In the future, the multiagent systems class will cover the theory and algorithms, while a second class ("Distributed Programming") will cover the topics of Servlets, RMI, SOAP, WSDL, UDDI, WSDL, Workflow, OWL, OWL-S, and any other technologies that form part of the Semantic Web vision. We make this separation because there is not enough time in one semester to cover both topics and because there are many students interested only on the software engineering aspects of building simple systems and not on the complex mathematics and algorithms often required for building complex multiagent systems. For example, the last time these classes were offered, the multiagent systems class had 11 students while distributed programming had 26. We see this change as the normal evolution of a technology as it moves from research to application, becoming more complicated as it acquires all the associated technologies needed to solve real-world implementation problems.

3.2. A Unifying Notation for Multiagent Systems

The recent interest in mechanism design [7, 9, 12] within the multiagent community is due largely in part to the recognition that it provides an efficient, and up to now missing, formal framework for describing the problem faced by a designer of a multiagent system. To summarize, the mechanism design framework defines a set outcomes O and a set of types T where each agent i has a utility function $u_i(o)$ over all outcomes $o \in O$ that it tries to maximize and a type t_i which is a secret known only to the agent. A social choice function $f(\boldsymbol{t}) \rightarrow o$ is defined to be a mapping from the set of all agent types $\boldsymbol{t}$ to an outcome o. The goal of a mechanism designer is to come up with some mechanism M that

is a mapping from the set of actions taken by the agents to an outcome ($M(\boldsymbol{a}) = o$) such that the mechanism implements the *social choice function* f. That is, the mechanism M must one such that the agents will maximize their utilities if they take actions $\boldsymbol{a}$ and $M(\boldsymbol{a}) = f(\boldsymbol{t}^*)$ where $\boldsymbol{t}^*$ are the true types of all the agents. The outcome is often composed of the state of the system s along with a set of payments $\boldsymbol{p}$ for each agent.

The appeal of this formalism lies in the social choice function which serves to succinctly describe the desired emergent behavior of the system. The formal frameworks presented in Weiss' and Wooldridge's textbooks only describe the single agent; they fail to describe the global behavior which is, after all, the reason one builds a multiagent system. Mechanism design is also appealing because it has been studied by economists who have developed useful mechanisms. For example, if we assume that the agents have quasilinear utilities over the outcome and a sum of money they might get and that they do not mind revealing their true types then a payment formula such as the Vickrey-Clarke-Grooves (VCG) mechanism tells us exactly how much to pay each agent so that its dominant strategy will be to tell the truth about its true type.

The main drawbacks of mechanism design also stem from the fact that it is borrowed from economics. Namely, it assumes that agents are selfish, perfectly rational, and value money. Most of the solution mechanisms also assume that agents have quasilinear utility functions over the outcome and a payment, and that agents do not mind revealing their true types. These assumptions hold for some multiagent systems but not for all. Still, we propose that the basic framework can be extended in order to cover all systems.

Figure 3 shows how we organize the space of multiagent problems using the mechanism design notation along with traditional agent notation. There are two dimensions: the x-dimension represents whether or not we know the agents' decision function δ_i, the y-dimension represents whether or not we know how the collective actions of the agents will be mapped into an outcome. We use these to define a two-dimensional space and place various standard multiagent problems within this space.

The bottom left quadrant represents all the problems were we already know how the agents will behave and we know what the outcome for a vector of actions will be. Since everything is known there is no problem for the multiagent designer to solve in this case.

The bottom right quadrant corresponds to problems where we don't know how the agents behave but we do know the outcome that will result from their actions. The RoboCup problem, for example, fits this definition: we don't know how the agents behave (that is the problem the designers must solve) but we do know the rules of the simulator so we know the exact probability distribution over what will happen given the agents' actions. As such, we place RoboCup in the bottom right quadrant. Distributed constraint optimization and satisfaction problems [26], such as distributed graph coloring, the N-queens problem, and sensor networks [19], also reside in this quadrant. In these problems we are given a global utility function, typically something like "avoid constraint violations", and must determine how the individual agents must behave (δ_i) in order to maximize the global utility. Depending on how the problem is stated, we are sometimes also given some constraints on how much information is available to each particular agent which is just a way of limiting the space of possible agent behaviors.

The top left quadrant represents systems were we already know the agents' behavior functions and we must determine how to map from their collective actions to a final outcome. This quadrant is inhabited almost exclusively by systems with selfish agents. By definition, a selfish agent acts to maximize its utility and in almost all cases these utilities are known a priori. For example, in coalition formation each agent wants to be in the coalition that brings it the highest utility, in the mailman problem [16]—an instance of the task allocation problem—we are told that each mailman wants to minimize the total distance it must travel. The problems in this quadrant require us to come up with the mapping from joint actions to an outcome. A subset of this quadrant is populated by systems that obey the typical mechanism design assumptions, namely that the agents have quasilinear utilities over the outcome and a payoff ($u_i = v_i(o, t_i) + p_i$) and that the mechanism results in an outcome that consists of a state s and a payoff for each agent($\{s, \boldsymbol{p}\} = M(\boldsymbol{a})$). For these systems we can use well-known solutions such as VCG, marginal cost, Shapley value payments, and others, as long as we are willing to accept any limitations they might have (for example, most of them lack budget balance). As such, Figure 3 places the solutions offered by mechanism design within the larger problem space of multiagent systems.

While this space does help us categorize multiagent problems, it still does not completely capture their differences. Specifically, the RoboCup problem is hard because predictions on the outcome rely on all of the agents' actions, including those of the opposing team which the designer cannot predict. That is, the space as shown assumes that all decision functions that are not known will be implemented by the designer of the system. A better depiction would show the difference between systems where the designer controls varying number of decision functions, or varying parts of the mechanism function.

Still, we continue to use Figure 3 as our new map for the study of multiagent systems. While the space is currently only sparsely populated, we believe that all multiag-

ent problems can be placed within this space. Furthermore, the space shows us how we might develop a unifying formal notation for describing all multiagent systems, from cooperative planning systems in uncertain domains to combinatorial auctions. Such a notation could be explained in the first couple of days of classes and then re-used throughout the semester for the introduction of new problems. We are developing this notation for our next class. Figure 3 is only the first step towards our goal of a unifying notation.

4. Summary

We have provided a summary of the lessons we have learned after teaching a graduate multiagent systems class six times over five years. We find that a hands-on approach is best for helping students understand emergent dynamics and distributed algorithms—an approach that is greatly facilitated by the use of RoboCup and NetLogo. We have also pointed the way towards a unifying notation for describing multiagent problems. We hope to have this notation fully developed for use in our next class.

References

[1] F. Bellifemine, A. Poggi, and G. Rimassa. Developing multiagent systems with a FIPA-compliant agent framework. *Software: Practice and Experience*, 31(2), 2001.

[2] T. Berners-Lee, J. Hendler, and O. Lasilla. The semantic web. *Scientific American*, May 2001.

[3] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. Technical report, W3C Working Group Note 11, 2004. `http://www.w3.org/TR/ws-arch/`.

[4] C. H. Brooks and E. H. Durfee. Congregating and market formation. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 96–103, 2002.

[5] P. Buckle and R. Hadingham. FIPA-OS: FIPA everywhere, October 1999.

[6] P. Buhler and J. M. Vidal. Biter: A platform for the teaching and research of multiagent systems' design using robocup. In A. Birk, S. Coradeschi, and S. Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V. LNCS/LNAI Lecture Notes Volume 2377*, pages 299–304. Springer-Verlag, Berlin Heidelberg, 2002.

[7] R. K. Dash, N. R. Jennings, and D. C. Parkes. Computational mechanism design: A call to arms. *IEEE Intelligent Systems*, 18(6):40–47, Jan./Feb. 2003.

[8] J. Delgado. Emergence of social conventions in complex networks. *Artificial Intelligence*, 141:171–185, 2002.

[9] J. Feigenbaum and S. Shenker. Distributed algorithmic mechanism design: Recent results and future directions. In *Proceedings of the 6th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 1–13. ACM Press, New York, 2002.

[10] H. J. Goradia and J. M. Vidal. Building blocks for agent design. In P. Giorgini, editor, *Agent-Oriented Software Engineering*, pages 153–166. Springer-Verlag, 2004.

[11] P. J. Modi, W.-M. Shen, M. Tambe, and M. Yokoo. An asynchronous complete method for distributed constraint optimization. In *Proceedings of Second International Joint Conference on Autonomous Agents and MultiAgent Systems*, July 2003.

[12] N. Nisan and A. Ronen. Algorithmic mechanism design. *Games and Economic Behavior*, 35:166–196, 2001.

[13] H. V. D. Parunak, S. Brueckner, and J. Sauter. Synthetic pheronome mechanisms for coordination of unmanned vehicles. In *Proceedings of the First Intenational Joint Conference on Autonomous Agents and Multiagent Systems*, pages 448–450, Bologna, Italy, 2002. ACM Press, New York, NY.

[14] M. Pollack and M. Ringuette. Introducing the tileworld: experimentally evaluating agent architectures. In T. Dietterich and W. Swartout, editors, *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 183–189, Menlo Park, CA, 1990. AAAI Press.

[15] M. Resnick. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. MIT Press, 1997.

[16] J. S. Rosenschein and G. Zlotkin. *Rules of Encounter*. The MIT Press, Cambridge, MA, 1994.

[17] S. Sen. Believing others: Pros and cons. *Artificial Intelligence*, 142(2):179–203, December 2002.

[18] J. M. Vidal. An incentive-compatible distributed recommendation model. In *Proceedings of the Sixth International Workshop on Trust, Privacy, Deception, and Fraud in Agent Societies*, pages 84–91, 2003.

[19] J. M. Vidal. A method for solving distributed service allocation problems. *Web Intelligence and Agent Systems: An International Journal*, 1(2):139–146, 2003.

[20] J. M. Vidal and P. Buhler. Teaching multiagent systems using robocup and biter. *The Interactive Multimedia Electronic Journal of Computer-Enhanced Learning*, 4(2), 2002.

[21] J. M. Vidal and P. Buhler. Using robocup to teach multiagent systems and the distributed mindset. In *Proceedings of the 33rd ACM Technical Symposium on Computer Science Education*, pages 3–7, 2002.

[22] N. Vlassis. A concise introduction to multiagent systems and distributed AI. Informatics Institute, University of Amsterdam, Sept. 2003. `http://www.science.uva.nl/~vlassis/cimasdai`.

[23] G. Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[24] U. Wilensky. NetLogo: Center for connected learning and computer-based modeling, Northwestern University. Evanston, IL, 1999. `http://ccl.northwestern.edu/netlogo/`.

[25] M. Wooldridge. *Introduction to MultiAgent Systems*. John Wiley and Sons, 2002.

[26] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem: Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):673–685, 1998.