Multiagent Reactive Plan Application Learning in Dynamic Environments

By

Hüseyin Sevay

B.S.Co.E, University of Kansas, Lawrence, Kansas, 1991
M.S.E.E, University of Kansas, Lawrence, Kansas, 1994

Submitted to Department of Electrical Engineering and Computer Science
and the Faculty of the Graduate School of the University of Kansas
in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Prof. Costas Tsatsoulis
Committee Chair

Prof. Arvin Agah

Prof. Susan Gauch

Prof. Douglas Niehaus

Prof. Stephen Benedict
Department of Molecular Biosciences

Date dissertation defended

Abstract

This dissertation studies how we can build a multiagent system that can learn to execute high-level strategies in complex, dynamic, and uncertain domains. We assume that agents do not explicitly communicate and operate autonomously only with their local view of the world. Designing multiagent systems for real-world applications is challenging because of the prohibitively large state and action spaces. Dynamic changes in an environment require reactive responses, and the complexity and the uncertainty inherent in real settings require that individual agents keep their commitment to achieving common goals despite adversities. Therefore, a balance between reaction and reasoning is necessary to accomplish goals in real-world environments. Most work in multiagent systems approaches this problem using bottom-up methodologies. However, bottom-up methodologies are severely limited since they cannot learn alternative strategies, which are essential for dealing with highly dynamic, complex, and uncertain environments where convergence of single-strategy behavior is virtually impossible to obtain. Our methodology is knowledge-based and combines top-down and bottom-up approaches to problem solving in order to take advantage of the strengths of both. We use symbolic plans that define the requirements for what individual agents in a collaborative group need to do to achieve multi-step goals that span through time, but, initially, they do not specify how to implement these goals in each given situation. During training, agents acquire application knowledge using case-based learning, and, using this training knowledge, agents apply plans in realistic settings. During application, they use a naïve form of reinforcement learning to allow them to make increasingly better decisions about which specific implementation to select for each situation. Experimentally, we show that, as the complexity of plans increases, the version of our system with naïve reinforcement learning performs increasingly better than the version that retrieves and applies unreinforced training knowledge and the version that reacts to dynamic changes using search.

Acknowledgments

This dissertation would not have been possible without the guidance and support of my advisor, Prof. Costas Tsatsoulis. I deeply thank him. My mother and father believed in me as long as I know myself. I thank my sister and her family. They have been a great source of inspiration for me. I am grateful for the support of my entire extended family. My aunts, uncles, and cousins have lent their support to me for more than a decade and a half. During my stay at the University of Kansas, which became my home away from home, I have been fortunate to have many wonderful friends. I thank them all from the bottom of my heart. I shared so much with them that is so hard to put in words. I thank my professors who have encouraged me during my long journey. I also thank the ITTC staff. I am very grateful for all the support I received from them over the years.

Dedicated to my mother and father . . .

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The goal of this dissertation is to investigate multiagent problem solving in complex, highly dynamic, and uncertain environments from the perspective of how a group of autonomous agents can learn to apply high-level reactive multiagent plans to achieve shared goals in situated scenarios. Due to the distribution of control and reasoning, the multiagent systems (MAS) approach in Artificial Intelligence (AI) is well-suited for solving problems in such complex domains, and this distribution enables agents to react to dynamic external events as they collaborate to attain their long-term team goals.

Building decentralized solutions to problems that require reactive responses within a collaborative framework in complex, dynamic, and uncertain environments presents two major challenges. First, since complex environments with continuous states and actions have very large search spaces, at the single agent level, the solution methodology must address the problem of how to reduce these large search spaces. At the team level, it must address the problem of how to enable autonomous agents to collaborate efficiently and coherently. Second, the solution methodology must enable agents operating in real-world domains to handle the noise inherent in their sensors and actuators and the uncertainty in the environment.

In agent-based systems, solutions are built upon the basic action capabilities of individual agents. Therefore, in multiagent settings, the implementation of high-level strategies reduces to the coordinated sequences of basic actions of collaborating agents. In this dissertation, we refer to the data structures that implement high-level strategies as *plans*. In a complex environment, a team of agents may need a set of such plans to achieve their long-term goals. To be reactive to dynamic changes, a team must also be capable of generating quick responses to critical events happening in an environment. Therefore, a balance between deliberative and reactive behavior is essential to solving problems in complex, dynamic, and uncertain domains.

The question then becomes: *How can we build a multiagent system that can execute high-level strategies in complex, dynamic, and uncertain domains?* We may consider three possible answers to this question. First, we can build a system that successively executes plans chosen from a static library. Second, we can build a system that can learn its strategies from scratch. Third, we can describe the strategies symbolically at an implementation-independent level and have the system learn how to implement the necessary implementation-level details under varying conditions to be effective in situated scenarios. In our work, we pursue the third approach.

Most work in the multiagent learning literature has treated the challenge of building team-level strategies as a Reinforcement Learning (RL) problem. RL generates a strategy that emerges from an incrementally modified sequential decision memory over many training iterations. In complex domains, bottom-up learning techniques require practical convergence to provide stable policies, and, by their nature, they do not bound the search problem beyond rewarding every decision according to its perceived value since they intend to discover policies. Moreover, they suffer from the exponential growth of the search space as a factor of the size of the input vector. Therefore, scaling bottom-up learning approaches to large search spaces is a very difficult problem. On the other hand, we hypothesize that top-down approaches can constrain the search space, resulting in a more effective method for learning in multiagent systems.

We approach the problem of enabling autonomous agents to operate in complex, dynamic, and uncertain environments not from the perspective of learning of strategies but from the perspective of having each member of a team learn how to fulfill its part in given high-level plans. We assume that users can provide high-level symbolic descriptions of strategies that are believed to be effective in an environment. These symbolic plans describe *what* needs to be done from each contributing agent's perspective in order to execute a strategy but not *how* each task is to implemented.

A plan is initially a high-level specification for the implementation of a strategy, and it is decomposed into an ordered list of steps each of which may require the collaboration of multiple agents to implement its goal. A plan step, in turn, defines a specific *role* for each collaborating agent. A *role* describes the necessary conditions for executing and terminating the responsibilities of each given agent from that agent's local perspective of the world. Since the sequence of actions required for each situation can vary, before any learning takes place, a plan step does not contain any implementation-specific details about what actions each collaborating agent needs to take to perform its own task in that plan step. Therefore, at the outset, a plan is only a high-level specification of a strategy whose implementation-level details need to be acquired from experience in a situated environment. To acquire these details, our approach uses learning.

Unlike systems that learn policies directly from experience in a bottom-up fashion, our system does not learn plans or policies. Instead, high-level plan descriptions constrain the search and each agent autonomously learns action knowledge in this focused search space from its own perspective of the environment. In our approach, learning is done using a combination of case-based learning and a naïve form of reinforcement learning. An agent uses case-based learning to *operationalize* its role in each step of a plan. It then uses a naïve form of reinforcement learning to decide which plan to pick for a situation and then to decide which specific implementation of a plan step to activate to implement the current plan step successfully in the current context. The case-based learning component helps to deal with noise in matching situations to implementations, and the naïve reinforcement learning helps with the uncertainty in the environment, enabling the agents to choose the effective implementations. By training in different controlled scenarios, each agent acquires action knowledge that enables it to adapt its role to specific situations. By applying plans in regular situations, the agent collects reinforcements based on the success and failure of its learned action knowledge. This reinforcement knowledge allows the agent to make progressively better choices about which plans to use and which plan implementations to execute.

We demonstrate our learning approach experimentally using the RoboCup simulated robotic soccer environment [Kitano et al., 1997a,b; Chen et al., 2002]. We describe the RoboCup environment in *Section 2.10.1*.

1.1 Motivation

Most existing multiagent learning systems that operate in complex, dynamic, and uncertain environments use reinforcement learning techniques where a policy or plan emerges from the interactions of the agents with the environment and the interactions of agents among themselves (e.g., [Stone and Veloso, 1999; Matarić, 1996]). Due to their bottom-up nature, these systems are difficult to scale up to larger and more complex problems [Stone and Sutton, 2001]. They are also naturally not conducive to learning multiple distinct high-level strategies [Matarić, 1991], since they depend on practical convergence to provide coherent behavior for the policy they are trained for. Instead of depending on emergence to bring out a successful policy, we believe that constraining the multiagent learning problem in a top-down fashion using symbolic plans and formulating the solution as one of reactive plan application learning has certain advantages over purely bottom-up policy learning approaches.

In a domain where humans possess expertise, it may be beneficial to involve the expert user in the solution process [Myers, 1996, 1997]. The user charts a number of symbolic high-level plans, and then the system learns the details of when and how to apply each given plan, since manually specifying all the necessary details for all possible scenarios would be a very arduous task. This approach would be in contrast with systems that discover their own policies within the context of less constrained learning problems as in RL. In addition, a symbolic approach allows for the explicit decomposition of a large task into manageable steps, since a multiagent strategy is usually composed of a series of actions by several agents. This breakdown, in turn, reduces the search space. Moreover, conditions for success and termination of each step can also be specified explicitly, such that a system can determine when a strategy step is completed and when it should be terminated.

Domains with continuous search spaces pose a difficult challenge for learning [Stone and Sutton, 2001; Smart and Kaelbling, 2000]. Even though a subset of the problem can have a manageable search space with a relatively small number of agents in the case of MAS, adding more agents into the problem tends to render the solution approach ineffective. This is, for example, true of Reinforcement Learning (RL) approaches, which assume discrete state and action spaces [Stone and Sutton, 2001; Sutton, 1996; Merke and Riedmiller, 2002].

In general, a learning algorithm either requires discretized domain features [Stone and Sutton, 2001; Smart and Kaelbling, 2000] or uses the domain features without discretization [Merke and Riedmiller, 2002] using a form of function approximation. However, in multiagent learning as opposed to single-agent learning in complex, dynamic, and uncertain environments, it is virtually necessary to discretize all continuous search spaces. Even with discretization, there are no multiagent learning algorithms with provable convergence that can learn a single, all-encompassing team strategy in the kind of environments we investigate in this dissertation. This is mainly due to the problem of the *curse of dimensionality*. Therefore, existing approaches concentrate on learning more constrained group strategies rather than learning strategies that would cover all agent behavior in a given multiagent domain. So, a high-level arbitration mechanism would be required for deciding when to activate which individually-learned strategy and when to terminate the currently active one.

The solution to this problem would be to have the system learn under what conditions to apply as well as terminate each multiagent strategy so that the multiagent system can automatically switch between behaviors. But that learning problem could prove as difficult as the problem of learning a multiagent strategy for a relatively large team of agents due to the exponential growth of the search spaces. Systems that could learn a single multiagent strategy have already been reported in the literature [Whiteson and Stone, 2003; Stone and Sutton, 2001; Kostiadis and Hu, 2001], but the problem of learning a single team strategy that covers all agent behavior still remains unsolved.

Symbolic learning techniques, however, do not suffer from the *curse of dimensionality* to the same extent as do emergent learning techniques since a symbolic approach can introduce top-down constraining of the search space. If it were possible to design a multiagent learning algorithm that could learn all necessary agent behavior for a dynamic and complex domain without needing any high-level specification of what to do, then there would be no need for explicit programming of plans or strategies. Even if the training phase took a relatively long time, having an automated learning system that can cover an entire domain would clearly be of great advantage. However, the state of the art has not reached this point.

With a symbolic learning approach it is possible to design strategies with clearly defined boundaries where preconditions and termination conditions of each strategy are explicitly known. The tradeoff is then the explicit representation that would be required for specifying plan preconditions and termination conditions. In addition, it would be natural for a symbolic approach to contain a higher-level reasoning mechanism for choosing and terminating roles in multiagent strategies during the lifetime of each agent in a team.

Most closely related research in the literature uses either a bottom-up learning approach [Parker and Blumenthal, 2002; Stone and Veloso, 1999; Werger, 1999; Luke et al., 1998; Luke, 1998; Balch, 1997b; Matarić, 1996] or does not address the problem of multiagent learning [Simon Ch'ng, 1998b,a; Bersano-Begey et al., 1998] or is limited to single-agent learning [Tambe et al., 1999; Merke and Riedmiller, 2002].

1.2 Approach

We based our learning approach on the idea of *learning by doing* [Anzai and Simon, 1979], that is, learning from practice, and we established the benefits of our solution experimentally. As a testbed to demonstrate our methodology, we used the RoboCup Soccer simulated robotic soccer environment [Kitano et al., 1997a,b, 1995]. The soccer game provides a very rich multiagent environment that is both dynamic, complex, and uncertain.

Agents in the RoboCup simulated soccer environment are capable of only basic actions which need to be sequenced in order to provide meaningful behavior. The simulator provides a complex environment where player sensors and actuators are noisy. It also places strict limitations on agent-to-agent communication in terms of how frequently each agent can communicate using explicit messaging and how much bandwidth each message can use. Hence, these properties give rise to a realistic multiagent environment that is highly dynamic and unpredictable. Besides individual moves, soccer naturally requires collaborative behaviors involving multiple agents, and, for these behaviors to succeed, team activity needs to be tightly coordinated.

To implement our system to demonstrate the learning capability of our methodology, we manually implemented high-level individual skills based on the basic behaviors provided by the simulator. This hierarchical approach is akin to the *layered learning* in [Stone and Veloso, 1999; Stone, 1998], which used machine learning techniques.

To facilitate plan oriented multiagent behaviors, we divide each plan into steps. Each step specifies the roles that must be dynamically filled by agents at runtime. In a given scenario, each agent chooses the plan that best matches that scenario, and, subsequently, that agent assigns all agents involved in that scenario, including itself, their roles, and these roles remain fixed until the plan application terminates either with success or failure. However, each agent carries out its role assignments independently and does not share them with other agents it intends to collaborate with. Each agent has its own plans, and it selects which plan to execute and which role to take on in that plan autonomously.

Our learning approach is divided into *training* and *application* phases. Agents learn in both phases, but they acquire different types of knowledge in each phase. In the training phase an agent uses case-based learning to implement its role in a given plan step under the current scenario. This knowledge we call the *application knowledge* contains the sequence of actions the agent needs to execute in order to achieve the goal in the current step and under the conditions in which it is being trained. To determine this sequence of actions, an agent takes a snapshot of its current local environment and does a search in this environment according to the description in the current plan step. Although the scenario given to the search process is temporally static and cannot make accurate predictions of future states, the later application phase helps determine if the sequence of actions suggested by this local search will be useful. Since one implementation may not be sufficient, an agent acquires multiple possible implementations for each step.

In the application phase the agent selects and executes plans in collaboration with other agents in its group. As it applies plans to the current environment, each agent positively reinforces plans that succeed and negatively reinforces plans that fail. In each plan step, the agent positively reinforces cases that implement the current step successfully and negatively reinforces cases that fail to achieve the goal of that step. By continuous reinforcement, agents refine their selection of plans and plan step implementations.

Another feature of our approach is that we separate conditions external to the team from the conditions internal to it. Conditions external to a team are those the team does not have any direct control over, and conditions internal to the team are those the team can potentially control. At the least, each agent can affect its own state via its actions to affect the team internal state. To match plans to scenarios, we only use the conditions that are internal to the team. Then, to distinguish the finer details of each situation, we use the conditions external to the team to index each piece of application knowledge associated with a given role in a plan.

During training, agents act in their regular operating environment as they do during the application phase, except that the training environment is more controlled to allow agents to acquire application knowledge. During the application phase, each agent chooses an uninstantiated plan, assumes a role for itself and deduces roles for other team members. Then it attempts to apply its responsibilities arising from its chosen role to completion in each plan step. Coherence in the multiagent behavior comes from the reinforcement of plans and plan parts as well as the definition of each role in each given plan.

In summary, our experimental approach uses a symbolic case-based learning method combined with a naïve reinforcement method to implement reactive plan application learning in complex, dynamic, and uncertain environments. We show that our learning methodology improves the behavior of a team of agents in the case of each plan.

1.3 Contributions

This dissertation makes several contributions to the field of Artificial Intelligence (AI), specifically to Multiagent Systems (MAS):

- A methodology for constructing a *learning by doing* solution to complex multiagent problems in dynamic and unpredictable physical environments.
- An agent architecture that enables reactive and collaborative strategy selection and application in situated environments.
- A plan application approach that uses a combination of case-based reasoning and a naïve form of *reinforcement learning*.
- A symbolic representation method for specifying both high-level multiagent and single-agent plans and for storing application-specific knowledge in the form of cases.
- A method for physical domain situation matching for testing the applicability of plan preconditions and the satisfaction of plan postconditions.
- An unsupervised learning algorithm that incorporates case-based learning and a naïve form of reinforcement learning.
- A fully-implemented system that incorporates our multiagent learning methodology.

1.4 Dissertation Outline

The remaining chapters of this document are organized as follows:

- **Chapter 2 (Background)** is a survey of research in areas related to our work. This background work covers multiagent systems, planning, machine learning, case-based reasoning, and knowledge representation.

- **Chapter 3 (Methodology)** presents our methodology for multiagent reactive plan application learning. It describes our agent architecture, knowledge representation method, algorithms for multiagent reactive plan application learning in dynamic environments, and evaluation method.

- **Chapter 4 (Implementation)** provides the details of our implementation of the ideas we describe in this dissertation.

- **Chapter 5 (Experimental Results and Discussion)** presents and discusses the results of the experiments we conducted using the RoboCup soccer simulator.

- **Chapter 6 (Conclusions)** presents the conclusions of our work and suggests possible directions for future work.

◇

Chapter 2

Background

In this dissertation, we are interested in investigating how to enable a team of agents to work on shared goals by learning how to apply high-level multiagent plans in dynamic, complex, and uncertain environments. Dealing with such environments requires that agents have reactive behavior, and the goal-oriented nature of plans requires that agents be capable of accomplishing their long-term goals in the presence of challenges posed by the environment. The goal of learning in this dissertation is to provide a general mechanism for recognizing situations to which each agent in a cooperative group knows how to react in the service of the group's objectives.

Since this document focuses on learning in dynamic, complex, and uncertain multiagent environments, we start with a historical perspective on how problem solving in AI has evolved to this day. Then we present issues and work relevant to our research.

2.1 A Brief History of Problem Solving in AI

The field of Artificial Intelligence (AI) was born out of the motivation for creating general-purpose problem solvers that would exhibit some characteristics of human problem solving. In late 1950's, Allen Newell and Herbert Simon proposed the General Problem Solver (GPS), which introduced the seminal ideas of means-ends analysis and difference-finding between non-goal states and a goal state as fundamental to their approach to creating domain-independent systems [Newell and Simon, 1963].

At the tenth ACM Turing Award Lecture in October 1975, Newell and Simon described their general approach formulated as the *Physical Symbol System Hypothesis*, which stated that "[a] physical symbol system has the necessary and sufficient means for intelligent action." The physical symbol system hypothesis provided for symbols that could be arranged into expressions and a set of processes that operated on these symbol structures to produce other expressions in the course of solving problems [Newell and Simon, 1976]. Starting with these core ideas, AI took root as a research field, and, by early 1970s, new groundbreaking systems started to appear.

Developed initially for robot control, STRIPS demonstrated means-ends analysis by successively transforming its world model by applying operators defined in terms of their applicability conditions (preconditions) and effects on the world model (postconditions) [Fikes and Nilsson, 1971]. This approach to specifying operators in terms of preconditions and postconditions set a lasting standard, and STRIPS, as a complete system, motivated much of the subsequent work in planning and problem-solving. Its assumptions and limitations have served as new points for further investigation.

ABSTRIPS extended STRIPS by planning in hierarchies of abstraction spaces instead of at the level of state descriptions. Even though ABSTRIPS used the same operator definition and planning method as STRIPS, it created abstraction spaces by treating some preconditions as more critical than others during planning. So by treating each precondition in the order of its criticality to a problem, it successively refined its plan after finishing planning at each abstract level [Sacerdoti, 1974].

NOAH introduced the concept of partially-ordered action descriptions and non-linear planning. It used *critics* that added constraints to plans during planning to eliminate redundant operations and to order actions to avoid subgoal interactions. NOAH used *procedural nets* to define plans as a partially-ordered network of actions, and this hierarchical description of actions enabled NOAH to focus on the relevant parts of the problem. However, NOAH did not do any backtracking, and this limited the type of problems it could solve. NONLIN extended the non-linear planning ideas of NOAH further by adding backtracking and introduced *task formalisms* to hierarchically describe a domain.

MOLGEN, a hierarchical planner, introduced the idea of constraint posting. Constraint posting uses constraints to represent subproblem interactions explicitly [Stefik, 1981], and this technique has since become commonplace in AI planning. Planners such as SIPE [Wilkins, 1984] and DEVISER [Vere, 1983] incorporated limits on resources as part of planning and reasoned about them. DEVISER provided for the specification of time limits on goals and other activities so that the planner could reason about the time windows within which goals had to be achieved and how long certain conditions should be preserved [Vere, 1983]. SIPE also incorporated replanning in response to plan failures.

Case-based planners such as CHEF introduced the approach of retrieving existing skeletal plans applied in contexts similar to that of the new planning problem and then modifying the retrieved solution instead of planning a solution from scratch [Hammond, 1989, 1986]. The PRIAR system integrated the case-based planning approach with traditional planning approach [Kambhampati, 1990; Kambhampati and Hendler, 1992].

The Procedural Reasoning System (PRS) reasoned about problems in terms of the beliefs, desires, and intentions of the planner. Its primary goal was to integrate goal-directed planning with reactive planning to provide highly reactive behavior and yet goal-directed behavior in dynamic and uncertain environments. To accomplish this, it created a hybrid approach by interleaving planning and execution [Georgeff and Lansky, 1987].

In the 1980s, the realization that traditional AI planning approaches may not be well-suited for practical applications led to the development of reactive systems [Brooks, 1986; Firby, 1987; Agre and Chapman, 1987]. Some researchers went as far as to question the validity of the physical symbol hypothesis [Brooks, 1986] on which almost all work on AI was based. They suggested that intelligent behavior could be generated without requiring explicit symbolic representation and done so without the abstract reasoning of traditional AI systems, and that intelligence was an emergent property of complex systems driven by the interactions of problem solvers with their environments [Brooks, 1986, 1989, 1991]. This new wave of research has paved the way for AI systems that increasingly took into account the properties of real-world environments.

A large body of work sought to build upon the fundamental ideas in AI as the interest in the research community progressively shifted towards realistic domains. From the perspective of this dissertation, we can identify two major trends in the literature. First, the interest in creating modular solutions led to distributed problem solving (DPS) approaches in AI, exemplified by Blackboard systems and the Contract Net Protocol. Closely related to DPS, a new body of work also emerged in the form of multiagent systems, which, in general, deals with enabling a number of autonomous agents to solve problems that are difficult or inefficient to solve using single-agents alone [Bond and Gasser, 1988; Chaib-draa et al., 1992; Durfee and Rosenschein, 1994]. Second, moving away from toy problems towards more realistic domains required that traditional *plan-then-execute* approach be modified into an interleaved planning and execution approach [Ambros-lngerson and Steel, 1988; Wilkins, 1984]. It became apparent that, to cope with the complexities of real-world environments, AI systems needed to be reactive to cope with dynamic changes and be deliberative to be able to stay committed to their long-term goals. Therefore, systems that interleave planning with execution strive for timely responses rather than solution optimality. After all, in dynamic, complex, and uncertain environments, defining what is optimal is as complex as finding an optimal solution.

2.2 Dynamic, Complex, and Uncertain Multiagent Domains

Early AI work had concentrated on centralized, single-agent solutions, but later work looked for ways to distribute computing and control to multiple agents. In addition, the type of domains considered included more realistic domains that involved multiple agents instead of a single agent, and complex, dynamic, and uncertain worlds instead of static, predictable worlds.

Particularly, the assumption known as the “STRIPS assumption,” which held that the world will change only in ways defined by the postconditions of the operator that is being applied to the current world model, no longer held in dynamic environments. In a dynamic and complex environment, it is difficult for an agent to know exactly how the world will change as a result of its actions. Moreover, when there are other agents in an environment, it is no longer the case that only a single entity is capable of acting to cause changes in that environment. Therefore, a goal-oriented agent has to be able to react to unexpected events caused by both the environment and other agents, while managing to pursue its long-term goals in the face of the uncertainties and complexities of that environment.

Unlike in traditional planning, the low-level actions of agents in real physical domains are durative, and their effect is uncertain. Situated and embodied agents know about the world through their sensors and affect changes in the world through their actuators, both of which are imperfect and limited. Sensors provide noisy information about the changing aspects of the world, and the information they provide is partial, hiding potentially important aspects of the world from the agents, giving rise to the problem of *hidden state*. Effectors are also noisy, and, therefore, they do not cause changes in the state of the world in precisely the intended ways. Moreover, actions may fail to bring about the changes they were executed for. For example, an object being detected at 10 meters may not necessarily be at exactly 10 meters away, and a command for turning a robot 10 degrees may not necessarily execute an exact 10-degree turn. And, if, for example, the robot fell in a ditch and is stuck, attempting a turn to presumably avoid an object will not accomplish that goal.

Dynamic real-world domains also require that problem solvers reason about resources and the temporal nature of actions and changes while pursuing their goals by reacting and deliberating in the presence of other collaborative agents. This means that agents need to be adaptive, and preprogrammed approaches tend to be limited in providing adaptability. Therefore, learning can provide the means for adaptability in dynamic, complex, and uncertain environments.

2.3 Deliberative Systems

Central to deliberative systems is reasoning about goals, the state of the world, and the internal representations of the planner. Contrary to *reactive* systems, which map situations to actions, deliberative systems generate plans to accomplish their goals. Therefore, the main concern in deliberative systems is the time and space complexity of the algorithms used to generate solutions.

We can characterize problem solving in real-world domains as *action selection*, which happens in response to the question of "What to do next?" [Hexmoor and Nute, 1992].

Traditional AI planning systems deal with this problem by choosing an entire sequence of actions so that the initial state of the world can be transformed into the desired goal state by they direct application of those actions (e.g., [Fikes and Nilsson, 1971; Sacerdoti, 1975; Tate, 1977; Stefik, 1981; Wilkins, 1984; Erol et al., 1994]). Given a problem, these systems generate a complete plan before executing any of the actions of that plan. This approach to problem solving is driven by a set of basic assumptions:

- The world in which the planner operates is static.
- The agent that executes the primitive actions on the world state is the only entity that can cause the world state to change.
- Lowest-level or primitive actions that will potentially cause changes in the world state take zero time.
- The effects of such primitive actions are deterministic. That is, an action only brings about the changes that it intends to cause in the world state.

Even with these assumptions in place, problem solving is still hard due to combinatorial explosion. Because of this reason, AI systems face difficult challenges.

Combinatorial explosion: The search space for a problem increases exponentially in the size of the problem description [Tate et al., 1990; Georgeff, 1990] Therefore, the search problem in planning is, in general, intractable (NP-hard) [Chapman, 1987]

Search space reduction: Since the planning problem is NP-hard, we need ways to reduce the search space complexity. One possible reduction method involves reformulating the problem using abstractions so that not as many states need to be searched. Another method is to use heuristics to control the search so that only a subset of the search space need to be searched [Georgeff, 1990].

Scalability: It is also desirable that the algorithm produce solutions that can be efficiently computed in terms of time and space for problems of gradually increasing size in the given domain.

All of these challenges are common to both single-agent problem solving and multiagent problem solving. In the case of multiagent systems, however, the inherent problem of combinatorial explosion becomes more limiting for AI algorithms, since introducing a new agent into the problem exponentially compounds complexities of existing state and action spaces, as opposed to introducing, for example, a new primitive action.

2.3.1 HTN Planning

Hierarchical Task Network (HTN) planning differs from STRIPS planning in the way it represents planning goals. In STRIPS, a goal is a symbolic propositional expression that denotes the desired change in the world. In HTN planning, STRIPS goal specification is replaced by tasks and their associated task networks that specify how to accomplish those tasks. This formulation of planning problems is a way to reduce search [Erol et al., 1994].

There are three types of tasks in HTN planning. Goal tasks are analogous to STRIPS goals and represent properties that must be made true in the world. Primitive tasks can be executed directly. Compound tasks are made up of a number of goal tasks and primitive tasks. Both compound and goal tasks are non-primitive.

The input to an HTN planner consists of a task network that has been chosen to solve a given problem, a set of operators, and a set of *methods*. State and operator representation in HTN planning is same as in classical planning. An operator has a precondition and specifies the effects of applying a primitive action. A method represents how to perform a non-primitive task with constraints and contains a set of preconditions and a totally-ordered list of subtasks. The representation of states in HTN planning is also the same as in STRIPS.

HTN planning works by expanding non-primitive tasks into successively more specific subtasks using methods, eventually leading to primitive tasks before any execution can take place. When there are no more non-primitive tasks left from the initial task network to be expanded, the next step is to find a totally-ordered instantiation of all primitive tasks that satisfies the constrains in the initial task network. If such an instantiation is possible, then the resulting order of primitive actions is a solution to the original problem.

HTN planning also involves critics for handling functions such as task ordering constraints, resource contentions, and domain-specific planning guidance in order to eliminate problems during planning.

Even though task decomposition using hierarchical task networks facilitates an effective planning approach, a designer needs to formulate standard operating procedures in a domain and express those procedures in terms of methods and operators.

2.3.2 Heuristic Search Planning

Heuristic search planning (HSP) involves treating the planning problem as a heuristic search problem. A heuristic search planner automatically extracts a heuristic function from the STRIPS-style declarative problem representation, and it uses this heuristic function to guide the search as in traditional heuristic search scenarios [Bonet and Geffner, 2001a; Bonet et al., 1997]. The heuristic function is created by considering a relaxed version of the problem, where delete lists are ignored, thereby assuming subgoal independence. Even though this assumption is not valid in general, it was shown to be useful [Bonet and Geffner, 2001a].

Since solving the relaxed version of the original problem is still hard [Bonet and Geffner, 2001a], an approximation of the heuristic function is used, and the value of this approximate function is updated until its values stabilizes. Since HSP uses a heuristic to control search, it is a hill-climbing planner. At every step, one of the best of the children is selected for expansion, ties are broken randomly, and this process is repeated until the goal state is reached.

2.3.3 Soar

Soar is a general-purpose problem solver rooted in the Physical Symbol System Hypothesis [Newell and Simon, 1976], and it is implemented as a specialized production system. Soar represents all tasks in terms of heuristic search to achieve a goal in a *problem space*. A problem space consists of the set of operators that are applicable to a state and the new states that are generated by those operators. The long-term knowledge of Soar is encoded in productions. Productions provide both search control knowledge and procedural knowledge about how to achieve goals [Newell, 1990; Rosenbloom et al., 1993].

At each problem-solving iteration, Soar fires all matching rules in parallel and applies no conflict resolution. Whenever direct knowledge is available, a decision is made. If that is not the case, an *impasse* occurs, since problem solving cannot continue. To resolve the impasse, a subgoal is automatically created to get that knowledge. This automatic goal creation in Soar is called *universal subgoaling*.

Search control knowledge in Soar is expressed by *preferences*. A preference represents knowledge about how Soar should behave in the current problem space, and it is created by productions that add such data to the working memory of the production system. Preferences describe acceptability, rejection, and desirability (best, better, indifferent, worse, worst) about different aspects of the current problem.

The decision cycle of Soar contains two phases. In the *elaboration* phase, it fires all satisfied productions to generate all possibly generatable results including preferences. Then, in the *decision* phase, Soar imposes control on search using the data generated by the previous phase. In the second phase, Soar examines all generated preferences and makes a final decision about what to do. As a result of its decision, Soar may apply an operator, or it may create a subgoal if an impasse occurred.

Since Soar works mainly by creating subgoals, it employs a form of explanation based learning method called *chunking* to cache its prior problem-solving experience as generalized productions. When a decision cycle for a goal ends, a chunk is created to summarize the processing required in solving a subgoal, so that similar problems in the future can be solved more efficiently since no subgoaling will be needed [Newell, 1990; Rosenbloom et al., 1993].

2.3.4 BURIDAN

The BURIDAN family of planners follow a classical approach to planning but they incorporate uncertainty [Kushmerick et al., 1994]. BURIDAN plans assume that sensors and effectors can be faulty, and the agents can have incomplete information. This approach uses Bayesian conditional probability theory to deal with uncertainty. Actions in this system are described as probabilistic mappings between states. Instead of a unique successor state from a given state, BURIDAN associates a conditional probability to each state that can be reached from a given state based on the current action. Besides this probability specification, plans in this system are represented in the STRIPS style, with preconditions and effects [Kushmerick et al., 1994].

2.4 Reactive Systems

Contrary to traditional planning systems that generate plans to transform the world state into the desired goal state, purely reactive systems do not explicitly reason about long-term goals and do not generate plans. A reactive system is composed of a set of actions and a control framework that selects actions to respond to changes in an environment. Instead of producing complete plans to solve a problem, reactive systems match actions to situations to be able to produce timely responses to dynamic changes by switching between behaviors when necessary. Therefore, ideally, the response time and behavior switching time of a reactive system should not fall behind the changes in the environment that it is designed to respond to.

Another feature of reactive systems is that they do not have an explicit representation of their policy, since they do not generate plans and keep minimal state information. Rather, the sequence of actions a reactive system produces during its interaction with the environment emerges as the policy encoded in the control framework of the system.

Since timely response is most critical to handle dynamic changes, optimality is not the main issue in reactive systems. The goal is to produce "good-enough" solutions as quickly as possible. So, even if allowing more computations to take place would produce a better quality response for the situation at hand, the quality of the response must be traded off with reaction time, since late responses may be of little or no value. Even though timely response is critical in reactive systems, most well-known reactive systems do not have explicit support for real-time constraints [Adelantado and Givry, 1995].

In general, we can characterize a reactive system as one that does fast action selection. A reactive system executes a reaction that matches the current situation, and it continues to apply the same reaction until a change occurs in the environment that prompts the system to match a different reaction to the new situation. Therefore, as part of its control mechanism, a reactive system neither detects failures nor replans. However, in theory, failure detection or replanning knowledge can be encoded in the reactive plans of the system. Therefore, we can say that reactivity lies in the plans and not in the planning strategy of the system [Hexmoor and Nute, 1992].

2.4.1 RAP System

The RAP system was designed for reactive execution of symbolic plans, and it works as a reactive plan interpreter. In this system, a goal can be described in multiple levels of abstraction. Then the RAP system tries to execute each level of abstraction while dealing with dynamically-occurring problems [Firby, 1987, 1989, 1994].

In the RAP system, a task is defined by a Reactive Action Package (RAP). A RAP is a program that carries out a specific task in a context-sensitive fashion. So a RAP may describe how to achieve a goal in different contexts. A RAP symbolically defines the goal of a task and specific methods for achieving the goal of the task in different situations. In each method, a RAP contains a context description to identify when that method is applicable and a *task net* that describes the procedural steps to perform.

The RAP system executes its tasks by checking whether the selected task represents a primitive action. If so, that task is executed directly. If the task is non-primitive, then it looks up in its RAP library for a RAP that implements that non-primitive task. Then it checks the goal success condition of the retrieved RAP. If the goal success condition is satisfied, then the task is considered complete, and the system can run another task. If the success condition is not yet satisfied, it checks the retrieved RAP's method applicability conditions and selects the method with the applicable context description. Then it queues the subtasks of the chosen method for execution and suspends the task until its subtasks are complete. When all of its subtasks are complete, the RAP system checks the success condition of the suspended task. If the success condition is satisfied, then the system can work on another task. If not, then it repeats the whole process by selecting another method to try to achieve the goal of the initial task.

The RAP system also allows the representation of concurrent tasks. This involves introducing synchronization primitives into the task-specification language and a method for representing multiple possible outcomes of each subtask so that the non-determinism in an environment can be captured.

2.4.2 Pengi

Pengi [Agre and Chapman, 1987, 1989] is a reactive system that is based on the *plan-as-communication* view of plan use, as opposed to the *plan-as-program*, which classical STRIPS family of planners are based on. Pengi plays a video game called Pengo. In the plan-as-communication view, plans do not determine what agents do. Instead, plans are resources agents can use to decide what to do. Agre and Chapman believe that classical planning approaches create agents that attempt to control their environment, where in their plan-as-communication view of plan use, agents only participate in the environment but do not try to control it. So, Pengi constantly uses contingencies and opportunities in its environment to improvise its behavior for pursuing its goals. This kind of improvisation, however, only depends on reasoning about the current state of the world.

Pengi does not build or manipulate symbolic representations. Instead, it uses a *indexical-functional* representation that uniquely identifies entities in the world and their function. Pengi also does not construct any plans or models of its environment. When contingencies arise, Pengi aborts the routine it is executing and tries another action repeatedly until it works or until it gets an opportunity to try a more promising action. It interleaves different routines and uses its repertoire of actions in ways that may not be anticipated. This leads to a form of creative improvisation.

2.4.3 Situated Automata

In the situated automata approach of creating reactive agents, the operations of an agent are described declaratively by the designer. Then the declarative specification is compiled into a digital machine that can generate outputs with respect to its perceptions using its internal state. Even though the agent is described symbolically, the generated machine does not do any symbolic manipulations [Kaelbling, 1991; Rosenschein and Kaelbling, 1995]. Although reactive in its responses, agents created using situated automata do not have the ability to adapt to their environment beyond their initial design capabilities.

2.5 Behavior-based Systems

Behavior-based systems are based mainly on the Subsumption Architecture [Brooks, 1986], but they also share features with the purely reactive systems. A behavior-based system has a collection of behaviors that it executes in parallel as each module gathers information about the world through sensors. There is no reasoner or central control in a behavior-based system, and the integration of the system is achieved through the interactions of the behaviors among themselves and mostly through the environment. Typical goals of behavior-based systems require both switching among behaviors as well as maintaining behaviors, and the system can have state and other representations to enable such capability [Matarić, 1992].

The behaviors of a behavior-based system are built bottom-up, starting with the most essential basic behaviors such as obstacle avoidance and moving to a designated location. Then more complex tasks are added keeping with the intended overall behavior of the system. However, the more complex behaviors are not built in terms of simpler behaviors, but they interact with the more basic behaviors to produce behaviors that no single behavior module is capable of exhibiting [Matarić, 1992].

Since behavior-based systems are both situated and embodied, they are inherently reactive in the sense that they produce timely responses to changes. However, behavior-based systems are classified separately from reactive systems in general, since behavior-based systems do not map a single action to a given situation but rather adjust the interaction of multiple behaviors so that the combination of behaviors generates the more complex behavior needed to implement the goal of the system [Matarić, 1992]. However, some researchers categorize behavior-based systems as reactive systems.

One of the main design challenges of behavior-based systems is action selection. Since behavior-based systems do not employ central control, selection of the appropriate behavior for a situation requires arbitration among the available behavior modules. Well-known examples of arbitration mechanisms include *spreading activation* where hand-tuned thresholds are used to trigger actions [Maes, 1989] and voting schemes [Payton et al., 1992].

2.5.1 Subsumption Architecture

The *Subsumption Architecture* is based on three principles: (1) that intelligent behavior can be generated without using explicit representations of states and goals as done in traditional symbolic AI systems; (2) that intelligent behavior can be created without high-level reasoning; (3) that intelligence is only an emergent property of a complex system that stems from its interactions with the environment and not from any preprogrammed behavior. In addition, situatedness and embodiment are key to Subsumption. Situatedness refers to an agent operating in a real environment but without creating or updating a model of that environment. Embodiment refers to the agent operating in a real physical environment [Brooks, 1985, 1989, 1991].

One of the guiding principles of this architecture is that an intelligent system must be built incrementally by building upon layers of what are themselves complete modules. The Subsumption Architecture does not depend on a symbolic representation of the world or any symbolic manipulation for reasoning. Brooks suggests that the world be used as its own model, rather than building models in software. The claim is that there is no need to explicitly represent the world or the intentions of an agent for that agent to generate intelligent behavior.

The Subsumption Architecture is layered, where each layer contains a set of augmented finite state machines (AFSMs) that run asynchronously. In addition to a regular finite state machine, AFSMs contain registers and timers. Registers store the inputs arriving into AFSMs and timers help the AFSMs change state after a designated amount of time passes. There is no central control in the Subsumption Architecture because each AFSM is driven by inputs it receives from the sensors or the outputs of other AFSMs. Perception is connected to action directly without symbolic reasoning in between. Each layer in this architecture specifies a pattern of behavior such as avoiding obstacles or wandering, and, since the AFSMs are connected to each other through their inputs and outputs, this exchange is viewed as message passing. The messages from other AFSMs are saved in the (input) registers of an AFSM. The arrival of such messages or the expiration of the timeout value of the timer internal to the AFSM prompts a state change. Each layer in the Subsumption Architecture is connected to other layers via a *suppression* and an *inhibition* mechanism. During suppression, the output of AFSMs connected to a given AFSM are blocked for a predetermined duration. During inhibition, the output of an AFSM is blocked for a predetermined period. The outputs of some of the AFSMs are directly connected to the actuators on the host robotic system.

In an example given in [Brooks, 1991], a robot is made up of three layers, where the lowest layer is responsible for avoiding obstacles, the middle layer is responsible for wandering around, and the top layer is responsible for trying to explore by attempting to reach distant locations. By itself, the lowest layer is capable of avoiding colliding with objects in the world. The *wander layer* generates a random heading to follow at regular intervals. The lowest layer treats this value as an attraction force towards that direction and adds it to the repulsive force computed based on sonar readings. The architecture then uses the result from this computation to suppress the behavior of the lowest layer, which avoids obstacles. The net effect is that the robot moves in the direction intended by the randomly generated heading, but it will also avoid colliding with obstacles; therefore the behavior that results at the wander layer subsumes the behavior of the lower layer. Similarly the top (exploration) layer suppresses the behavior of the wander layer, and makes corrections to the heading of the robot while it is avoiding obstacles so that the robot eventually reaches the target location.

2.5.2 AuRA

AuRA is a hybrid architecture for robot navigation [Arkin and Balch, 1997]. AuRA is based on the schema theory, where independently acting motor behaviors concurrently contribute to the generation of desired behavior. Schemas are represented as potential fields. At each point in a potential field, a behavior computes a vector value to implement the purpose of its schema. Behaviors are assembled together to produce more complex behaviors, called behavior assemblages, by combining the vector values generated by several schemas. For example, the GOTO behavior assemblage is made up of MoveToGoal, Wander, AvoidObstacles, and BiasMove schemas. That is, there is no arbitration among the schemas that make up more complex schemas since all schemas have the same goal of contributing to navigation. In addition, the behaviors are not layered.

AuRA uses uses a deliberative planner. A plan sequencer translates the navigation path determined by its spatial reasoner into motor schemas that can be executed. Once reactive execution begins, the deliberation component becomes inactive, unless there is a failure in the reactive execution of the current plan due to, for example, lack of progress. When a failure occurs, the hierarchical planner is called to replan the failed reactive portions of the current plan.

2.6 Hybrid Systems

Hybrid systems combine the advantages of reactive systems and deliberative planning systems by incorporating them in a three-layer architecture that consists of a reactive execution module, a deliberative planner, and a layer that links the reactive and deliberative layers [Gat, 1998]. This layer may be designed to generate reflective behavior, or it may be a simpler layer that allows communication of the reactive and deliberative components with no specialized high-level behavior. The general focus of hybrid systems has been in practical reasoning rather than optimality, which is difficult to study in complex systems with multiple interacting agents. While the low-level reactive module ensures that the agent can survive in an environment and respond to critical dynamic events, the high-level deliberative module plans actions for long-term goals [Gat, 1998].

2.6.1 Practical Reasoning, BDI, and PRS

The Procedural Reasoning System (PRS) is a system for executing and reasoning about complex tasks in dynamic environments [Georgeff and Lansky, 1987]. PRS is based on the *Belief-Desire-Intention* (BDI) model of agency [Georgeff et al., 1999]. The BDI model aims to facilitate practical reasoning in rational agents [Bratman et al., 1988; Rao and Georgeff, 1995].

PRS has four types of data structures: (1) a database of *beliefs* about the world, (2), a set of goals (or *desires*) to be realized, (3) a set of declarative procedures or plans called *Knowledge Areas* (KAs) that describe how to perform tasks to achieve goals or react to dynamic changes, and (4) an *intention structure* that contains all currently active KAs selected to achieve current reactive or deliberative goals. The system is run by an interpreter.

A plan has several parts. It has a trigger or an invocation condition that specifies under what circumstances it can be considered for execution. A trigger is usually represented in terms of an event (for example, the "make tea" intention may be triggered by the condition "thirsty" [d'Inverno et al., 1997]). A plan has context description or precondition that specifies under what conditions the execution of a plan may start. A plan can also specify a maintenance condition, which specifies a set of conditions that must be true as the plan is executing.

The body contains the recipe of how to accomplish a certain task and is a tree structure that describes the flow of operations. Instead of representing a sequence of primitive actions as in traditional planning, it represents possible sequences of subgoals that must be achieved. A plan can be described in terms of temporal conditions as well as control constructs such as condition, iteration, and recursion. In the special case of a primitive action, the body is empty.

Besides domain-specific KAs or plans, PRS also has meta-level KAs that are used to manipulate the beliefs, desires, and intentions of the system. Meta-level KAs can use domain-specific information.

It is possible for a BDI agent to have multiple desires that are "mutually incompatible." Although real-world environments can potentially change very quickly, agents cannot afford to reason about the environment continuously, and therefore, they have to commit themselves to certain goals with the *intention* of carrying them out eventually.

The interpreter works by cycling through a set of processes. It observes the world and its internal state and updates the beliefs of the system. If the trigger and context conditions of a plan are satisfied by the updated beliefs, that plan is selected as a potential goal. After determining all plans that match the current events reflected in the beliefs, the interpreter selects one to place on the intention structure for eventual execution. Finally, the interpreter chooses one of the intentions stored in the intention structure to execute.

Since PRS agents do not plan from first principles at runtime, all plan details need to be created manually at design time. PRS implementations have been used in handling malfunctions in the reaction control system of the space shuttle [Georgeff and Ingrand, 1989] and in real-time reasoning [Ingrand and Coutance, 1993].

2.6.2 Cypress

Wilkins et al [Wilkins et al., 1995] describe a domain-independent framework, Cypress, which is used for defining reactive agents for dynamic and uncertain environments. Unlike traditional planning systems, Cypress does not make the assumption that agents have perfect domain knowledge, since dynamic environments are unpredictable. As other reactive planning systems, Cypress also tries to balance reactive behavior with deliberative behavior. It also addresses *failure recovery*, such that agents have the capability to overcome difficulties that arise because of failures.

Cypress is a hybrid system that comprises a generative planner, a reactive plan execution subsystem, and a reasoner for uncertainty. Cypress can handle problems that arise during execution by replanning new alternatives using its generative planner. The interesting part is that it does not have to stop its plan execution. The parts of the current plan that are not affected by the problem continue to be executed while replanning goes on for the problematic parts.

Each agent is made up of three components: an executor, a planner, and a planning library. The executor is active at all times. It monitors the environment for events that require actions to be performed, and it carries out those actions. The executor has three main responsibilities. It runs plans that are stored in the agent's planning library, calls on the planner to generate new plans for achieving goals, or asks the planner to modify its plans that lead to problems during execution. The planner in Cypress is not a low-level planner; rather it plans down to a level of abstraction above the primitive level only; subsequently the executor expands that plan to lower-level actions according to the current runtime conditions. The planner does not plan to the smallest detail possible, because without runtime information, it is not possible to determine a priori what set of low-level actions will best fit the current situation. Therefore the level of detail in the planner is up to the level that it can reason about in advance, and no more.

2.6.3 ATLANTIS

The idea of planning with complete world information and reacting purely based on stimuli from the environment has been at odds with each other [Gat, 1993, 1992]. Gat offers a resolution of this issue by stipulating that the issue at heart is how the internal state information is used towards the achievement of goals.

Reactive architectures such as the Subsumption Architecture [Brooks, 1986] (See *Section 2.5.1*) do not store any internal state information. However, as Gat contends, as long as the stored internal state information is used in predicting the environment, saving state is advantageous over not doing so. Specifically the idea is that an internal state should be maintained at an abstract level and should be used to guide the actions of agents and not control them directly.

ATLANTIS deals with online planning, which is problematic for three reasons. The first problem is that complete planning is time-consuming; hence in a dynamic setting, an agent can miss to fail to respond to changes fast enough ("Oncoming trucks wait for no theorem prover." [Gat, 1993]). The second problem is that the classic planning approach requires a complete world model, which is infeasible in complex environments. Finally the third problem is that, while planning, the world may change in ways that invalidate the plan being produced. Gat proposes that the first problem can be solved by planning in parallel with dealing with contingencies. Gat stresses that the remaining two problems arise due to how the stored internal state is maintained, particularly when the stored state information does not reflect the realities of the environment. Moreover the problem is not with the contents of what is stored but with the predictions implied by the stored information. For example, in the case of soccer, an agent can predict the position of the ball incorrectly, if it depends on relatively old information stored about the previous positions of the ball.

Gat suggests that the solution to the prediction problem is to make predictions at a high level of abstraction. The reason is that high level abstractions will still be valid even if there are small changes in the world state. However, the sensor data is used to fill in the gaps of such abstract representations. However, abstract world models can sometimes be wrong. Gat stresses that this does not usually pose a problem if the abstractions are used for guiding behavior and not for controlling them. The remaining capability that humans possess in dealing with failures of expectation due to incorrect abstractions is the ability to recognize those failures and recover from them.

The ATLANTIS system incorporates these ideas to enable the classical planning approach to deal with real-time control problems. The ATLANTIS architecture consists of three components. The controller is a purely reactive system, which is responsible for controlling the actuators according to the information received from the sensors. The sequencer is responsible for controlling the computations of the controller, and the deliberator is responsible for maintaining the world model. The controller embodies a set of basic behaviors similar to the ones used in other architectures such as the Subsumption Architecture [Brooks, 1991; Matarić, 1996], namely avoiding obstacles, following walls, etc. Then these basic behaviors are used to construct more complex behaviors. However each basic behavior also incorporates the ability to detect failures so that the agent can invoke a contingency plan to recover from the failure. The ATLANTIS architecture has been applied in simulating the navigation behavior of a single robot, whose task is to collect objects and bring them to a home location in an environment with static obstacles. The system retains information about the location of obstacles so that future attempts avoid obstacles already recorded in memory as long as the world remains static.

2.6.4 IRMA

The Intelligent Resource-Bounded Machine Architecture (IRMA) is a practical reasoning system based on BDI. Its goal is to enable resource-bounded agents to exhibit rational behavior using means-ends reasoning for planning and a filtering mechanism in order to constrain the scope of deliberation [Bratman et al., 1988].

The purpose of plans is both to produce action as well as to constrain deliberation. The fundamental tenet of the IRMA approach is that a rational agent should be committed to its intentions. Plans are used both to help focus means-end reasoning as well as to help filter out options that are inconsistent with current intentions of the agent.

The IRMA architecture has a plan library that stores the procedural knowledge of an agent. It explicitly represents an agent's beliefs, desires, and intentions. It also has an intention structure to keep track of the plans that the agent intends to complete The architecture has four major components: a means-end reasoner, an opportunity analyzer, a filtering mechanism, and a deliberation process. The means-end reasoner suggests subplans to complete structurally partial plans adopted by the agent. The opportunity analyzer monitors the dynamic changes in the agent's environment and proposes further plan options in response to those changes. Once all options have been generated by the means-end reasoner and the opportunity analyzer, IRMA uses one of its filtering mechanisms, namely the compatibility filter, to check whether the suggested options are compatible with the current plans of the agent. The options that survive filtering are then submitted to the deliberation process. Finally the deliberation process considers the available options, and weights competing the ones that may have survived compatibility filtering against each other to decide which intention to adopt next.

The second type of filtering mechanism of IRMA is the filter override mechanism. The filter override mechanism knows about the conditions under which some portion of exiting plans need to be suspended and weighed against another option. This mechanism operates in parallel with the compatibility filtering mechanism. The deliberation process is not affected by the filter override mechanism. By working in parallel with the compatibility filtering mechanism, it allows the agent to reconsider options that got filtered out by the compatibility filter but still triggered the agent to override the decision of the compatibility filter due to opportunities that might have arisen or other sensitivity that the agent has to the current situation that requires it to reconsider filtered out options. Then the agent reconsiders its current intentions that are incompatible with the options that triggered a filter override. The IRMA architecture has been evaluated in one-agent experiments using different deliberation and filtering strategies in the Tileworld simulation testbed [Pollack and Ringuette, 1990].

2.6.5 InteRRAP

InteRRaP is layered architecture for resource-bounded agents aims to provide reactivity, deliberation, and cooperation. It follows the BDI approach for representing an agent's goals, knowledge, and mental state [Jung and Fischer, 1998].

The InteRRaP architecture has a world interface, a control unit, and a knowledge base. The world interface allows perception and action as well as communication. The control unit has a hierarchy of three layers, the behavior-based layer, the local planning layer, and the cooperative planning layer. The agent knowledge base is also divided into three hierarchical layers corresponding to those in the control unit. The first layer, the world model, stores the beliefs of the agent, representations of primitive actions and patterns of behavior. The second layer, the mental model, represents the knowledge an agent has about its own goals, skills, and plans. The top layer, the social model, contains beliefs about other agents' goals to be used for cooperation.

The purpose of the behavior-based control layer is to react to critical situations and handle routine procedural situations. So, the knowledge stored at this layer consists of two types of behaviors: hardwired condition-action rules that implement the reactivity needed to respond to critical situations, and pre-compiled plans that encode the procedural knowledge of the agent. These behaviors are triggered by events recognized using the world model. The local planning layer enables the agent to deliberate over its decisions. It uses both the world model and the mental model, and it generates plans for the agent. The highest control layer, the cooperative planning layer, extends the functionality of the local planning layer to implement joint plans with other agents in the environment such that a group of agents can cooperate and resolve their conflicts. Besides using the world model and the mental model, the cooperative planning layer uses the social model of the agent knowledge base to plan joint goals and communicate with other agents for cooperation.

The architecture also limits the lower layers from accessing the knowledge base of higher control layers. So, for example, the local planning layer can access the world model, which is the knowledge base corresponding to the behavior-based layer, but the behavior-based layer cannot access either the mental model or the social model of the agent. This is done to reduce the reasoning complexity at the lower control levels of the architecture.

InteRRaP implements three types of functions. The belief revision and knowledge abstraction function maps an agent's current perception and beliefs to a new set of beliefs. The situation recognition and goal activation function (SG) derives new goals from the updated beliefs and current goals of the agent. The planning and scheduling (PS) function derives a new set of commitments based on the new goals selected by the situation recognition and goal activation function and the current intention structure of the agent.

All three layers of the control structure implement a SG and PS function, and they interact with each other to produce the overall behavior of the agent. If, for example, the PS function at level i cannot handle a situation, S, it sends a request for help to the SG function at level $i+1$. Then the SG function at level $i+1$ enhances the description of the problematic situation, S, and reports back the results to the PS function at level i so that the situation can be handled.

In addition to activation requests to upper layers of the control architecture, InteRRaP provides for commitment posting to lower layers so that the PS function modules can communicate their activities. For example, the local planning layer can use partial plans from the cooperative planning layer and take into account the commitments of the upper layer. InteRRaP has been tested in an automated loading dock domain.

2.6.6 TouringMachines

[Ferguson, 1992a,b] describes a hybrid architecture for resource-bounded agents operating in realtime environments called the TouringMachine. The TouringMachine architecture has its inspiration in the Subsumption Architecture [Brooks, 1986] (See *Section 2.5.1*), but, unlike in Subsumption, the TouringMachine architecture uses symbolic representation and keeps internal state.

The input to the architecture is handled by a perception subsystem, and the output of the architecture is conveyed to actuators using an action subsystem. An internal clock defines a fixed period for generating inputs to the system using the perception layer and outputs from the system using the action subsystem so that inputs and outputs are synchronized. The architecture comprises three control layers that work concurrently and are able to communicate with each other to exchange control information and independently submit actions to be executed.

At the lowest layer, the reactive layer generates actions to enable an agent to respond to immediate and short-term changes in the environment. The reaction layer is implemented with a set of situation-action rules, and it does not do any search or inferencing to select which rule to execute. The reactive layer does not use an explicit model of the world and is not concerned about the consequences of the actions it proposes.

The planning layer is responsible for building and executing plans to implement the agent's well-defined achievement goals that define a start and a final state. The functionality of the planning layer is divided into two components, the *focus of attention mechanism* and the *planner*. The job of the focus of attention mechanism is to limit the the amount of the information that the planning layer has to store and manipulate by filtering irrelevant information for planning so that the agent can operate in a time-bounded fashion. The planner is responsible for constructing and execution a plan to achieve the agent's high-level goals. Since any layer of the control architecture can submit action commands every operating cycle, the planning layer interleaves planning and plan execution. Each layer can submit at most one action command per input-output cycle, and a mediating set of control rules resolves conflicts that might arise between the layers.

The TouringMachine architecture can have two types of conflicts between its control layers. One type of inter-layer conflict occurs if two or more layers attempt to take action for the same perceived event in the environment. Another type of conflict occurs if one or more layers try to take action to handle different events.

The control rules act as filters between the sensors and the control layers, and they are applied in parallel once at the start and once at the end of each operating cycle. *Censor rules* are applied at the start of a cycle, and they filter sensory information to the inputs of the three control layers. *Suppressor rules* are applied at the end of a cycle, and they filter action commands from the outputs of the three control layers. Both types of control rules are if-then-type rules.

Since the control rules are the only method of mediating the input to and output from the three control layers, each layer works transparently from each other. Therefore, even when one layer is unable to function, other layers can still continue to produce behavior. However, realistic domains require complex functionality, and the independence of the three layers is not sufficient to produce the type of complex responses required in dynamic and complex environments. Therefore the architecture has an inter-layer messaging system, so that each layer can assists others.

2.7 Cooperation, Coordination, and Collaboration

The most critical issue in enabling multiple agents to work together in a multiagent setting is the sequencing of individual actions such that the overall behavior is coherent [Jennings, 1996]. There are three levels of group activity that we can define using the terms *cooperation*, *coordination*, and *collaboration*. According to the Webster dictionary [Merriam-Webster],

- *cooperation* means "to act or work with another or others",
- *coordination* means "to put in the same order or rank" or "to bring into a common action, movement, or condition", and
- *collaboration* means "to work jointly with others or together especially in an intellectual endeavor".

There are also definitions available from the organizational development field [Winer and Ray, 1994]. *Table 2.1* gives a summary of these definitions from a set of perspectives that we adapted from [Winer and Ray, 1994, page 22] for multiagent systems.

As *Table 2.1* indicates, we refer to group activity in the loosest sense using *cooperation* and in the strongest sense using *collaboration*. We can then think of *coordination* to refer to activity that may be collaborative for short periods of time and cooperative at other times. However, these definitions of cooperation and coordination seem to be at odds with those used in AI literature.

	Cooperation	*Coordination*	*Collaboration*
Relationship	short-term informal relations among agents	longer-term, more formal relationships among agents	long-term, close relationship among agents
Goals	no clearly defined goal	common goal	full commitment to a common goal
Organizational Structure	no defined organizational structure	agents focus on a specific effort	a clear organizational structure is established
Planning	no planning	some planning occurs	comprehensive planning among agents
Communication	agents may share information	agents are open to communication	well-defined communication is established
Authority	still retain authority	still retain their individual authority	agents work together in the established organizational structure
Resources	resources are not shared	resources are shared	resources are shared

Table 2.1: Cooperation, coordination, and collaboration from the perspective of agent relationships, group goals, organizational structure, group planning, group communication, individual authority, and resource sharing (adapted from [Winer and Ray, 1994, page 22])

In AI parlance, it is usually asserted that coordinated activity does not necessarily imply cooperation as in the example of traffic where vehicles move according to commonly agreed rules. However, using the definitions of [Winer and Ray, 1994], we have to say that vehicle activity in traffic is only cooperative and not coordinated, and therefore we have to assert instead that "Cooperative activity does not necessarily imply coordination." We can also say that "Coordinated activity does not necessarily imply collaboration." The Webster dictionary definitions are consistent with this view, especially since the definition of coordination is associated with a common goal and that for cooperation is not.

Malone and Crowston define *coordination* from a different perspective:

> "Coordination is managing dependencies between activities." [Malone and Crowston, 1994]

They add:

> ". . . [E]ven though words like 'cooperation', 'collaboration', and 'competition' each have their own connotations, an important part of each of them involves managing dependencies between activities."

Even though this definition of coordination is not inconsistent with the definitions we adopted from [Winer and Ray, 1994], it is not distinguished from cooperation and collaboration in more specific terms.

[Doran et al., 1997] defines cooperation in a way that resembles the definition of coordination in [Winer and Ray, 1994]. It gives two conditions for cooperative activity: (1) That agents have a possibly implicit goal in common and (2) agents perform their actions not only for their own goals but for those of other agents. According to [Jennings, 1996], coordination is built upon commitments, conventions, social conventions, and local reasoning capabilities and is needed for three reasons:

1. There are interdependencies among agent actions.
2. There are global constraints that must be satisfied.
3. No one agent has the capability, nor the resources, nor the information to solve the problem.

2.8 Realtime AI

Realtime AI approaches attempt to bridge the gap between reactive AI systems and hard realtime systems. Compared to traditional deliberative AI systems, both reactive and hybrid systems provide some sense of real-timeliness due to their ability to generate relatively fast responses to dynamic events. However, they do so without any guarantees about how long it may take for an agent to respond to an event. Hence these approaches can be categorized as soft realtime. For instance, even if an agent produces a *correct* response, r, at time $t + \tau$ to an event, e, that occurred at time t, the deliberation or reaction period τ may be too long for that otherwise correct response to be effective or meaningful at time $t + \tau$, since the world may have changed meanwhile. If event e required a response within a time period of κ, where $\kappa < \tau$, then response r would be considered a failure.

Interestingly, as AI approaches consider more realistic domains and realtime systems get more complex, AI and realtime fields are becoming more interdependent. AI systems are in need of realtime properties, and realtime systems are in need of AI techniques . Therefore, besides being able to operate under *bounded rationality* [Simon, 1982], real-world intelligent agents have to be able to operate under *bounded reactivity* [Musliner et al., 1993] as well.

Bounded rationality says that an agent's resources place a limit on the optimality of the agent's behavior with respect to its goals. Bounded reactivity says that an agent's resources place a limit on the timeliness of the agent's behavior.

Hard realtime systems guarantee that execution deadlines will be met; otherwise catastrophic failure may occur. Therefore, hard realtime systems are not necessarily fast-responding systems [Stankovic, 1988]. Reactive systems, on the other hand, aim to produce continual fast response to dynamically occurring events in an environment. However, they do not make any guarantees about timeliness.

The responses of an intelligent system can be characterized in terms of three factors, completeness, timeliness, and quality. Completeness says that a correct output will be generated for all input sets. Timeliness says that the agent produces its responses before specified deadlines. Quality describes the quality and the confidence of the generated response [Musliner, 1993].

2.8.1 Anytime Algorithms for Learning and Planning

There are two main characteristics of an anytime algorithm. First, the algorithm can be suspended any time with an approximate result, and, second, it produces results that monotonically increase in quality over time. Anytime algorithms describe a general category of incremental refinement or improvement methods, to which reinforcement learning and genetic algorithms also belong [Grefenstette and Ramsey, 1992]. The major issue in anytime algorithms is the tradeoff between solution generation time and solution quality [Korf, 1990].

Grefenstette and Ramsey describe an *anytime learning* approach using genetic algorithms as a continuous learning method in dynamic environments [Grefenstette and Ramsey, 1992]. The goal of the system is to learn reactive strategies in the form of symbolic rules.

The system is made up of a learning module called SAMUEL that runs concurrently with an execution module.

The system has a simulation model of the environment, and the learning module continuously tests the new strategies it generates against this simulation model of the domain and updates its current strategy. The execution module controls the interaction of the agent with its environment, and it also has a monitor that dynamically modifies the simulation model based on the observations of the agent. Learning continues based on the updated strategy of the system. The anytime learning approach was tested in a two-agent cat-and-mouse game [Grefenstette and Ramsey, 1992]

The anytime planning approach in [Zilberstein and Russell, 1993] starts with low-quality plans whose details get refined as time allows. This approach was tested in a simulated robot navigation task with randomly generated obstacles.

[Ferrer, 2002] presents an anytime planning approach that replaces plan segments made up of operator sequences around failure points with higher-quality ones that are generated quickly, progressively replacing larger segments to increase the plan quality.

[Briggs and Cook, 1999] presents an approach for domain-independent anytime planning that has two phases for building plans. The first phase involves an anytime initial plan generation component that generates a partial plan using a hierarchical planner. When the deliberative planner is interrupted, a reactive plan completion component completes the plan using a forward-chaining rule base.

[Haddawy, 1995] presents an anytime decision-theoretic planning approach based on the idea that an anytime decision-making algorithm must consider the most critical aspects of the problems first. This approach uses abstraction to focus the attention of the planner on those parts of the problem that have the highest expected utility. This is called the *rational refinement property*. That is, it is not good enough to impose the constrains that (1) the planner can be interrupted any time and the expected utility of the plan never decreases, and (2) that deliberation always increases the expected utility of the plan. The third needed property is the rational refinement property. Actions are represented by conditions and probabilities, and uncertainty is represented using utility function over outcomes (maximum expected utility).

2.8.2 Realtime Heuristic Search

Realtime search methods enable the interleaving of planning and execution but they do not guarantee finding the optimal solution [Ishida, 1998]. The A* algorithm is a type of best-first search in which the cost of visiting a node in the search graph is determined by the addition of the known cost of getting to that node from the start state and the estimated heuristic cost of reaching the goal state from that node. Thus, this cost function provides an estimate of the lowest total cost of a given solution path traversing a given node.

A* will find an optimal solution as long as the heuristic estimate in its cost function never overestimates the actual cost of reaching the goal state from a node (the admissibility condition). The assumption with classic search algorithms such as depth-first, breadth-first, best-first, A*, iterative-deepening A* and the like is that the search graph can be searched completely before any execution has to take place. However, the uncertainty in dynamic environments and the increased computational costs stemming from the complexity of the environment renders the classic search approach impractical. Therefore a new search paradigm is necessary where the agents can interleave execution nth search since they cannot always conduct search to completion due to the complexity and the uncertainty involved in predicting future states of the problem space. Moreover the goal of search is frequently the optimization of the overall execution time spent in reaching the goal state from the start state in real time. Hence the interleaving of execution with search is imperative.

Real-Time A* (RTA*) and Learning Real-Time A* (LRTA*) are algorithms that enable an agent to implement this interleaving of execution and search [Korf, 1990] . The RTA* algorithm expands a node in the search graph and moves to the next state with the minimum heuristic estimate, but it stores with that node the best heuristic estimate from among the remaining next states. This allows RTA* to find locally optimal solutions if the search space is a tree. The LRTA* algorithm on the other hand stores the best heuristic value among all next states in the current node instead of the second best heuristic estimate, and with repeated trials that may start from different start states, the algorithm improves its performance over time since it always minimizes cost when it moves to the next state. As such, given an admissible heuristic function, the heuristic values of each state in the search graph eventually converges to its actual value giving rise to an optimal solution.

Even if realtime search is suited for resource-bounded problem solving, the intermediate actions of the problem solver are not necessarily rational enough to be used by autonomous agents. Moreover, they cannot be applied directly to multiagent domains since they cannot adapt to dynamically changing goals [Ishida, 1998]. An extension of LRTA* to non-deterministic domains has been proposed by Koenig [Koenig, 2001]. A variation of LRTA* has also been used to implement an action selection mechanism by considering planning as a realtime heuristic search problem where agents consider only a limited search horizon [Bonet et al., 1997]. This action selection mechanism interleaves search with execution but does not build any plans.

2.8.3 Realtime Intelligent Control

Cooperative Intelligent Real-time Control Architecture (CIRCA) is a real-time control architecture [Musliner et al., 1993]. It merges the ideas of AI with real-time systems to guarantee timeliness. The main strength of CIRCA is that it reasons about its own *bounded reactivity*, that is, deadlines provide an upper bound to how fast the system should produce a response. CIRCA trades off completeness with precision, confidence, and timeliness, which is the most important concern in real-time domains, as for example, airplane control systems and patient monitoring in intensive care units.

The CIRCA architecture is made up of parallel subsystems. The AI subsystem (AIS) in cooperation with the Scheduler subsystem are responsible for deciding which responses can and should be guaranteed by the the real-time subsystem (RTS). The RTS implements the responses that are guaranteed by this decision. The guarantees are based on worst-case performance assumptions produced by the system by running simple test-action pairs (TAPs), which are known to have worst-case execution times. The TAPs are run by the RTS. The AIS reasons about the results of these runs executed by the RTS, and in cooperation with the Scheduler searches for a subset of TAPs that can be guaranteed to meet the control-level goals and make progress towards the task-level goals of the system.

2.9 Multiagent Systems

There are two major goals of multiagent systems. The first is to enable autonomous contribution to the solution of problems so that there is no single point of failure in a system. The second is to enable the solution of problems that cannot be solved by a single agent or would be very difficult or impractical to do so. So, in general, multiagent systems provide modularity in both dealing with the problem space as well as with resources [Sycara, 1998]. A great deal of work in multiagent systems deals with robot control that are situated in actual physical environments or simulation .

2.9.1 Issues in Multiagent Systems

As the research trend in AI moves towards realistic domains and practical applications, it is becoming apparent that researchers must deal with a challenging set of issues in multiagent systems. In this section, we will briefly list a set of issues that relate to this thesis. So we will consider multiagent systems that are designed to work on common goals. A set of similar issues has been listed in [Sycara, 1998].

Problem Decomposition: How a problem is divided into parts affects how a group of agents can contribute to the solution of that problem. In a totally distributed agent network, it may be possible to assign tasks to individual agents such that, after each is done solving its own part, the solution to the entire problem can be constructed . Agents may be able to negotiate and bid on what to do based on their capabilities. In highly dynamic environments, however, an a priori division of work may be necessary due to limited communication opportunities and lack of reliable communication.

Cooperation, Coordination, Collaboration: While working autonomously, each agent needs to be related to other agents in its group with which it shares goals. The selection of the type of relationship among agents depends partially on the particular multiagent system's design but also on the domain. For example, in box pushing or soccer domains, individual agents must share a collaborative relationship to achieve tasks (e.g., [Riedmiller and Merke, 2003; Parker and Blumenthal, 2002; Stone and Sutton, 2001; Mahadevan and Connell, 1992]).

Communication: In many real-world environments, communication among agents is limited in terms of both bandwidth and message-exchange opportunities. Despite this limitation, agents still need to stay aware of each other and their environment in order to cooperate, coordinate, or collaborate with each other. Communication can be explicit, by passing messages, or it can be implicit, based on social conventions. Agents may negotiate among themselves to collect information they need to proceed with their assigned tasks and reason about what to do next.

Commitment: Commitment is key to plan success in complex domains [Bratman et al., 1988]. Continuous reaction to external events does not necessarily contribute to the progress needed to attain goals. Instead, agents need to keep their commitments to the plans they intend to carry out as they adjust to the dynamic conditions that may otherwise be prohibitive to achieving their goals.

Resource Management: Each agent needs to reason about its own local resources, and a team of agents must reason about their shared resources to make the best use of them while solving problems and overcoming difficulties [Bratman et al., 1988].

Role Assignment: Given a multiagent task, how parts of that task are assigned is critical for the success of the group of agents in achieving that task. However, role assignment may imply some form of communication among agents.

Representation and Reasoning: Representation of multiagent problems plays an important part in how a group of agents can reason about their common goals in terms of their actions and knowledge. Also, in dynamic environments, it is advantageous for agents to be capable of recognizing opportunities that may enable them to attain their goals more efficiently .

Learning: Learning is fundamental to AI systems. Some learning approaches enable systems to work more efficiently using learned knowledge, and others enable them to acquire new knowledge. In dynamic and complex environments, learning can play a critical part in gradual improvement of a system since it facilitates adaptability.

Among the issues a multiagent learning system has to deal with, successful collaboration is at the heart of solving shared problems. The level of coordination or collaboration required to solve a problem may however vary from domain to domain. In mobile robot systems, for example, it is imperative that each agent coordinate its actions with others such that it does not impede them.

Most work in multiagent learning has so far used Reinforcement Learning (RL). In standard RL, an agent learns which actions to choose next based on the feedback or reward it receives from the environment it is situated in. A RL problem is defined by a set of discrete states, a set of discrete actions, and a set of reward signals. An agent may receive both immediate and delayed rewards towards the achievement of a goal. By exploring the state space using its actions, guided by the reward signals it receives after executing each action, the agent can construct a policy to be able to operate in that environment. Delayed reinforcement signals are critical, since they help guide the search through the state space based on the future effects of current actions [Kaelbling et al., 1996].

RL methods can be divided into two groups: (1) model-free approaches, and (2) model-based approaches. Model-free approaches, such as Q-learning [Watkins and Dayan, 1992], do not require a model of the domain in order to learn. Instead the domain knowledge encoded in the reward function guides the learner towards finding optimal policies by learning a value function directly from experience. In model-based approaches, on the other hand, agents learn a value function as they learn a model of the domain. A model of the domain is one that makes a prediction about what the next state and next reward will be given a state and an action performed in that state [Sutton and Barto, 1998; Kaelbling et al., 1996; Wyatt, 2003]. However, in real-world domains, predicting the next state and action is impractical. For this reason, RL approaches adapted to real-world and realistic domains favor model-free approaches (e.g., [Stone and Sutton, 2001; Riedmiller and Merke, 2003]).

2.9.2 Practical Issues in Multiagent Learning

Learning systems in complex domains face a set of challenges both in the single-agent and the multiagent case, but, in the multiagent case, these challenges are exacerbated further by the interdependencies among agents.

Large Search Spaces and Discretization: In real-world complex domains, state and action spaces are continuous and hence practically impossible to search exhaustively. Even after discretization, state and action spaces can still be very large. Therefore, search spaces need to be reduced to make problem solving in such domains possible. Discretization is problematic because too coarse a discretization will blur critical differences, and too fine a discretization will make generalization difficult and may even be prohibitive for search due to the size of the search space [Smart and Kaelbling, 2000]. If, on the other hand, no discretization is done, then there still needs to be a method that allows exploration within the search spaces of interest. The use of neural networks as function approximators in Reinforcement Learning, for example, allows inputs to have continuous values. However, function approximators suffer from the problem of *hidden extrapolation*, that is, they do not make valid predictions for all queries [Smart and Kaelbling, 2000].

Credit/Blame Assignment: In the action selection framework, the credit/blame assignment problem deals with attributing a value that indicates the relative strength of positive and negative contribution of an action to achieving a goal. This contribution can also be temporal. In multiagent systems, this problem is compounded since achievement of goals requires actions from different agents that are frequently interdependent on the behavior of other agents.

Shifting Concepts: In single learner domains, the only agent interested in improving its behavior is the learning agent itself. In multiagent learner domains, however, the concepts each agent learns are potentially interdependent on the behavior of other agents, which are themselves in the process of learning, hence the concept that has to be learned from each agent's perspective shifts constantly. Therefore, reaching convergence for a large group of agents is very difficult.

Local View of the Environment: Since agents operating in real-world domains cannot have a global view of the environment, local decisions of each agent have to contribute to the achievement of the global common goal. Meanwhile, each agent faces the problem of *hidden state*, since it cannot always sense its local environment fully.

Noise and Uncertainty: Agents embodied in complex domains suffer from noise both in sensing and acting. Imperfect sensors do not provide exact data, and imperfect actuators do not execute commands with deterministic precision. In addition, an agent cannot predict what will happen next in the environment, since changes do not only originate from the current agent, unlike the case in traditional planning systems. Since noise and uncertainty can potentially have adverse effects on learning.

Precondition, Action, Operator, and Plan Learning

[Matarić, 1997b] presents a RL approach that addresses the problems of very large state spaces and credit assignment using a behavior-based approach. A behavior is goal-driven control mechanism for achieving or maintaining specific goals. Matarić uses *shaped reinforcement* to take advantage of domain knowledge in order to convert intermittent feedback into a continuous error signal. To do this, Mataric uses two types of reinforcement, heterogeneous reinforcement functions and *progress estimators*. Heterogeneous reinforcement is derived from achieving specific behaviors, which may be subgoals in complex tasks. Progress estimators evaluate progress towards a goal. Since progress estimators are knowledge-based, they decrease sensitivity to noise in correlating behaviors to conditions. They also terminate behaviors that are detected not to make progress. The RL learning approach has been tested in a group foraging task that uses behaviors for safe wandering, dispersion, resting, and homing. The set of conditions to consider is also provided, so that the learning of which conditions should trigger which behaviors is feasible.

The Candidate Elimination Method Learner (CaMeL) [Ilghami et al., 2002] is a supervised learning method for learning method preconditions in HTN planning (See *Section 2.3.1*). CaMeL uses plan traces as input to its learning algorithm .

[Benson, 1995] presents a learning method whereby an agent can learn action models from its experience and also from its observing a domain expert. Instead of learning control policies directly, this system (a part of Teleo-Reactive Agent with Inductive Learning (TRAIL)) learns action models, and, using these learned action models, it computes reactive plans. Most of the learning occurs during execution of reactive plans that the learner creates.

TRAIL uses a continuous/durative action representation. This means that such actions can be interrupted. An action is represented by a teleo-operator (TOP). A TOP is very similar to a STRIPS rule, except that its preconditions must be maintained throughout the execution of the actions of the plan. In addition, side effects can occur any time during the execution of a rule, unlike add and delete lists in STRIPS that are incorporated into the world description once the execution of a rule ends. Benson reports that most learning occurs during when TRAIL is following a reactive plan. Reactive plans are formed based on TOPs, and the plan executor informs the learner whether the current TOP has been successful or not. Then the learner identifies positive (successful) and negative (unsuccessful) learning instances. These instances are then fed into an inductive logic programming learner. Since Benson's work involves symbolic reactive learning, however, it cannot be adapted to autonomous multiagent learning.

[Haigh and Veloso, 1998] presents ROGUE, an agent that integrates planning, execution, and learning. ROGUE learns from execution situation-dependent rules to improve planning. The task planner in ROGUE is built on the Prodigy 4.0 planning and learning system. The task planner generates and executes plans for multiple goals that might arrive asynchronously. ROGUE interleaves planning and execution so that it can detect failures and successes in order to respond to them. The learning component looks at the execution traces of the agent to determine in which situations the task planner's behavior needs to improve or change. Then it correlates the features of the environment with the situation and creates rules that allow the agent to recognize the same situations to predict and avoid failures later.

[Gervasio, 1990] presents an explanation-based learning (EBL) strategy for learning general plans in a planning approach that integrates classical planning and reactive planning. Determining a correct plan is done in two stages. In the first stage, a completable plan is constructed prior to execution. The plan may be incomplete, but there is a provable guarantee that the missing parts can be filled in during execution. So, the second stage is the completion of a plan that takes place during execution. With this approach, a priori planning is simplified, and perfect knowledge is not required since execution-time information will be used to fill in the missing details of the plan.

This approach identifies three types of planning subproblems for which achivability proofs can be constructed: (1) repeated actions, i.e., loops, and terminal goal values; (2) continuously-changing quantities and intermediate goal values, and (3) multiple opportunities .

This approach makes two modifications to standard EBL. It introduces variables, called *conjectured variables*, to be able to reason able deferred goals. It also introduces a completion step using special operators called completors. The value of conjecture parameters are those that are determined during execution. The job of the planner is to recognize these kinds of parameters.

The completors are responsible for finding suitable values for conjured variables during execution of a plan. Corresponding to the three types of subproblems, there are three types of completors, one for each type: (1) iterators that perform a particular action repeatedly until some goal is achieved, (2) monitors that watch a continuously-changing quantity to determine whether a particular goal value has been reached or not, and (3) filters that look for an object of particular type.

Using the given domain theory, EBL first produces an explanation for why a training example is an instance of the target concept using the domain theory, and then it generalizes the explanation. EBL does not learn new concepts but instead expresses the target concept in an operationalized manner. So EBL is a method that learns from single training examples. The downside is that an EBL system requires a complete domain theory to be able to build generalizations. The learned operationalization knowledge can later be used for search control.

Haynes and Sen applied multiagent case-based learning (MCBL) in a multiple-predator-single-prey domain for overriding default behavioral rules of individual agents. In this domain, the default behavior of an agent is to move closer to the prey, unless there is a case it previously learned that overrides this default behavior [Haynes and Sen, 1996a]. An agent learns a new case when its current case base or behavioral rules fail to predict the state the agent will be in after taking a certain action in the current state, and the learned case causes the agent not to select the same action when it finds itself in the same state. If an agent $A1$ learns a case $c1$ as a result of its interactions with another agent, $A2$, then $A1$ has to make sure that the $A2$ takes the same action it did when $A1$ learned $c1$. Otherwise, $A1$ has to modify its model of $A2$. The main assumption in this work is that agents have accurate perceptions of actions and states of all agents in the domain. The predator agents in the domain do not communicate explicitly.

[Brazier et al., 1995] presents a multiagent formal specification framework called DESIRE (framework for DEsign and Specification of Interacting REasoning components). They applied this framework to the real-world problem of electricity transportation management. This system has seven agents. One of the agents continuously receives and processes data about the network, including alarms and component status, and two diagnostic agents that diagnose faults. One of the diagnostic agents determines the general region of an electrical fault, and the other one uses model-based reasoning to determine and verify the cause of the failure. In this system the diagnostic agents cooperate in the sense that the general area of the fault indicated by the unsophisticated agent helps the sophisticated model-based agent to prune its search space.

[Garland and Alterman, 1996] and [Garland and Alterman, 1998] discuss multiagent learning through Collective Memory (CM) for autonomous agent coordination. The basic idea is that learning using CM will lead to efficient long-term behavior even if short-term behavior is suboptimal. Collective Memory refers to the collection of procedural problem-solving experiences that a group of agents working on a common goal acquire. Each agent maintains a CM of procedural knowledge that is implemented as a casebase of execution traces of cooperative problem-solving episodes. These episodes are used as plan skeletons in future problem-solving situations. When possible these problem-solving traces are analyzed and improved before saving into the casebase. Agents also maintain operator probabilities in another CM as a tree structure using COBWEB [Fisher, 1987]. This COBWEB tree is used to estimate the probability of successfully executing operators. If the same operator has been attempted before, an agent can use this past experience to determine success probability of that operator directly. Otherwise, the agent can use the COBWEB tree which clusters each operator according to its observable attributes to get an indirect success probability estimate.

Agents do not have special cooperation or coordination strategies. They have common top-level goals, and they use their own perspective to act. Then through the use of the CM, agents start building towards having a common perspective on how best to solve a problem. Agents do this by remembering past successful cooperation scenarios. Communication is considered an action, so it is included in the execution trace, but communication occurs only at designated points . So agents can learn to reduce the amount of communication, as they learn to reduce the use of other operators.

The CM approach was tested in the MOVERS-WORLD domain. The task in this domain is to move furniture and/or boxes from a house into a truck and vice-versa. This domain has specialized agents. Some are lifters and some are hand-truck operators. Agents with the same ability also differ in the value of their common attributes, e.g., strength.

If there are significant changes in the environment or the top-level goals of agents change, agents use their casebase of procedural knowledge CM. When this is not the case (no significant changes occurred in the environment or no relevant plan is found in the casebase), then the agent either adapts the current plan or plans from scratch. When planning from scratch, the agent uses the operator probabilities to guide the search [Garland and Alterman, 1996, 1998]

[Lee et al., 2002] discusses a technique that combines case-based reasoning and a crude reinforcement learning method called Learning Momentum (LM) to select behavioral parameters in a robot navigation task. In a similar work, [Likhachev et al., 2002] reports using case-based reasoning to learn what parameters to select for robot behavior assemblages in the AuRA robot architecture [Arkin and Balch, 1997]. The system starts with an empty case base, and it acquires cases whenever it does not already have a similar case in its case base. During application, each case is subject to adaptation of its parameters for further fine tuning of the parameters it stores for a given behavioral assemblage.

[Makar et al., 2001] proposes a multiagent extension to the MAXQ hierarchical reinforcement learning technique using explicit task structures . A task graph represents a particular task at several levels. Higher levels contain joint tasks, and the lowest level tasks are primitive tasks that can be directly executed. This abstraction hierarchy reduces the state space. This approach was tested in a trash collection task.

Learning in RoboCup Soccer

One of the earliest learning systems was by Littman, who used soccer as a domain in a two-player zero-sum Markov game [Littman, 1994]. Another early learning work by Matsubara et al. [Matsubara et al., 1996] introduces the robotic soccer simulation as a standard problem to study for multiagent systems. Matsubara et al use neural networks with backpropagation to teach agents whether they should shoot the ball or pass it to another teammate. They used a limited number of players in their experiments.

The learning system described in Stone's thesis [Stone, 1998] contains three distinct layers where each layer builds upon the the capabilities of the layer immediately below. These layers are: (1) ball interception, (2), pass evaluation, and (3) pass selection. The ball interception and pass evaluation layers use offline learning techniques, and the top layer, pass selection, uses an online learning technique.

In the lowest layer, ball interception, an agent learns an individual skill to take control of an incoming ball both for blocking shots from opponents and receiving passes from teammates. This layered was implemented both empirically using neural networks as well as analytically. Stone reports that both approaches worked equally well [Stone, 1998, Chapter 5].

In the middle layer, pass evaluation, an agent learns a multiagent behavior, since passing directly involves two agents, the sender and the receiver. This layer evaluates whether a pass to a teammate will be successful or not using C4.5 decision tree learning [Quinlan, 1993]. For a pass to be successful, the sender must execute a kick action correctly in the presence of opponents and the receiver must successfully receive the ball using its ball interception skill from the ball interception layer.

The middle layer introduces control functions called receiver choice functions (RCFs). The input to a RCF is the current state of world from an agent's perspective, and the output is description of what the agent should do with the ball.

The pass evaluation layer uses 174 features, 87 of which are described from the receiver's point of view and 87 of which are described from the passer's point of view. Since C4.5 decision trees can deal with missing features, an agent can still make a decision even if some features of the environment are currently missing, such as players that are currently unseen.

The top layer, pass selection, comprises learning a team behavior. By using its pass evaluation skill, the agent currently in possession of the ball has to choose which of its teammates to pass the ball to next, and the agent uses its pass evaluation skill to learn this skill. This layer is implemented using a reinforcement learning technique introduced by Stone called Team-Partitioned Opaque-Transition Reinforcement Learning (TPOT-RL) [Stone and Veloso, 1999; Stone, 1998].

TPOT-RL is a modified version of Q-learning. As in Q-learning, TPOT-RL also learns a value function to map state-action pairs to expected future rewards. The basis of the argument behind TPOT-RL is the need for quick generalization based on relatively rarely occurring learning opportunities when the learning agent has possession of the ball. To achieve this kind of generalization, the decision tree pass evaluation function learned at the middle layer is used as main component of the input representation to the TPOT-RL algorithm. In addition, each team member learning pass selection from its own perspective reduces the state space. The decision tree for pass evaluation reduces the input world description to a single continuous number, the confidence factor of the success of a pass.

In the soccer context, goals would receive the highest reward. However, since goals are relatively rare, TPOT-RL uses *intermediate reinforcement* based on how far the ball is advanced into the opponent's field. This intermediate reinforcement is applied within a fixed time window starting from the current state, and, because of this limited time window, rewards are temporal unlike long-term rewards in Q-learning. The rewards obtained in later states are then discounted based on how far into the future a rewarding state is reached. Therefore, the ball being passed back and later moved up the field towards the opponent's goal can result in a positive reinforcement.

The action space in TPOT-RL discretizes the passing action such that an agent only has 8 specific locations on the field to kick the ball to. Instead of kicking towards specific players, the discretization scheme presumes there will be teammate at the intended location of the pass.

The AT Humboldt system [Burkhard et al., 1998, 2000] is based on the Belief Desire Intension (BDI) architecture [Rao and Georgeff, 1995]. It uses a decision tree to decide which goal to pursue, but the set of goals it can choose come from a predefined planning library. The learning in AT Humboldt is limited to individual skills such as ball shooting and ball interception.

Parker and Blumenthal use genetic algorithms to co-evolve two separate populations in order to provide two specialized team members in a simulated box-pushing task [Parker and Blumenthal, 2002]. To enable efficient partnering of the most specialized individuals from the two separate populations, they use a punctuated anytime learning approach. Instead of choosing the most fit individuals from each population every iteration, this technique chooses the most fit individuals at regular intervals. In the interim generations, the most fit team members chosen from the last punctuated generations are paired with individuals from the opposite generation for fitness. Since the fitness function for a multiagent system has to recognize the best overall team, partnering one weak individual with a specialized individual would cause the learner not to recognize the specialized individual's strength. Therefore, using the most fit members from the last punctuated generation for fitness evaluation allows the genetic algorithm to evaluate the fitness of each individual with the most specialized partner from the other generation to find the best team for the cooperative task of box-pushing [Parker and Blumenthal, 2002].

Balch describes a simulated multi-agent system [1] where each agent learns a specialized role for playing soccer [Balch, 1997b]. This system uses groups of independent primitive behaviors to implement higher-level behaviors called assemblages. Since static assemblages are not adaptive, the system uses Q-learning to enable agents to pick the most appropriate behavior assemblage in the current situation. The work reported in [Balch, 1997b] assumes that sensors are perfect. A method for integrating reinforcement learning into behavior-based robot control is described in [Balch, 1997a].

Genetic algorithms have also been used to create soccer playing teams [Luke et al., 1998; Luke, 1998]. This work by Luke et al. divided the learning problem into two, one for moving each player and another for kicking the ball. At the high level, a simple set of rules controlled each player that made use of the learned moving and ball kicking behavior whenever appropriate. Players did not use communication and had no internal state due to the difficulty of including such capabilities in the genetic algorithms context within a complex domain such as soccer [Luke, 1998].

In recent years, the simulated keepaway game in soccer has been used to study how to scale up reinforcement learning to a complex, noisy, and highly dynamic environments [Kuhlmann and Stone, 2003; Stone and Sutton, 2001; Kostiadis and Hu, 2001]. In the keepaway game, one of the two teams tries to keep the possession of the ball as long as possible, while the opponent team attempts to get the ball. This work uses a subset of each team (usually 3 vs 2) for the keepaway task. A different set of state generalization techniques have been used in the reported work. [Whiteson and Stone, 2003] presents a concurrent layered learning approach following [Stone, 1998], also tested in the keepaway simulated soccer game. Instead of learning low-level behaviors first and constructing high-level behaviors using those low-level behaviors, the concurrent layered learning approach learns at both levels at the same time. The learning method used was neuro-evolution where genetic algorithms are used to train neural networks [Whiteson and Stone, 2003].

[Riedmiller and Merke, 2003] formulates soccer as multiagent Markov decision process

[1]The simulator used is different from the RoboCup soccer simulator.

(MMDP). This formulation involves a hierarchical approach to learning in the simulated RoboCup soccer environment, since agents need specific skills. Each such skill has a well-defined precondition and goal. By abstracting the low-level simulator commands away using skills, the search space is reduced. Instead of sequencing a list of instantiated primitive actions, instantiated skills need to be sequenced. Since each skill lasts a limited number of cycles, the number of decisions are much smaller than the number of decisions to be made for an entire game. The system learns a set of basic skills that include kicking the ball well, intercepting the ball, going to a position by avoiding obstacles, holding the ball, dribbling against an opponent. [Riedmiller and Merke, 2003] reports that learning team strategies with more than 4 players (2 attackers and 2 defenders) did not converge, since the state space grows exponentially with a bigger area of the field. The value function approximation for RL is done using a neural network.

[Wiering et al., 1999] applies reinforcement learning to learn strategies in simulated soccer teams using incomplete world models and a technique that combines CMACs and prioritized-sweeping-like algorithms. The simulator used is different from the RoboCup simulator, and teams have 1 to 3 players on each side, and there are impassable walls. In this simulator, ball possession is much simpler: if a player's body represented by a circle intersects that of the ball, the player controls the ball. The action set is also higher level than the RoboCup simulator. This is a model-based RL approach that does not use intermediate reinforcement. It only uses sparse reinforcement, namely goal scoring, as a chance to reward the successful team.

2.10 Simulated Robotic Soccer

In recent years, robotic soccer has been proposed as a benchmark problem to study AI and robotics in situated environments where traditional assumptions of AI no longer hold [Mackworth, 1993; Sahota, 1994]. Since the game of soccer naturally comprises multiple players that have to make autonomous decisions, it has also become a benchmark problem for studying multiagent systems [Kitano et al., 1997a,c,b]. To study algorithms in a more easily reproducible way for a complex and dynamic domain like soccer, a simulator has been in continual development and use by many researchers [Noda et al., 1998; Noda and Stone, 2001; Noda, 1995]

The RoboCup soccer simulator provides a realistic environment for studying many aspects of multiagent systems. Since the game of soccer is highly dynamic, cooperation among agents requires fine-grained coordination. However, limitations in agent capabilities and the adversarial and uncertain characteristics of the environment pose challenges for setting up and executing coordinated behaviors. Since the environment can potentially change very quickly, the agents must be able to react to those changes in service of their longer-term goals. At the same time, the complexity of the environment requires deliberation. That is, the agents must balance deliberation and reaction.

Meanwhile, each agent must also manage its resources to get optimal behavior from its actions. In the simulated soccer environment, we can speak of three major resources that must be managed well. First, an agent that is in possession of the ball must make good decisions as to what to do with the ball. Should the agent dribble the ball to a different location before passing the ball? Which player should the current agent pass the ball to? Second, agents have finite stamina, which gets expended every time the agent executes a move command. So, how much of its stamina can the agent afford to expend at a given time? Third, the agents can communicate in the simulation environment, but, similar to real physical robotic systems, this ability is very limited. When should an agent send a message and what can the message say in order to help with collaborative behavior?

2.10.1 Simulator Overview

As a discrete event simulator, *Soccer Server* provides a simulated robotic soccer client-server environment that is realistic and complex for multiagent problem solving [Noda et al., 1998]. An *agent* in this simulator soccer domain works as a *client* program that controls a single player. The client connects to the simulator through a UDP socket. The agent then uses this connection to send commands to the simulator and receive periodic feedback from the simulator using a specialized interprocess communicating language.

Each player receives periodic visual and self-awareness feedback from the simulator. The visual feedback arrives on during the first two cycles out of every three consecutive cycles. The self-awareness feedback, on the other hand, arrives at the start of every simulation cycle, and it includes the counts of each type of primitive command already executed by the Soccer Server on behalf a player and other information about its current state.

For controlling each player, the simulator offers to each client program a small set of action commands. Most of these commands are parameterized, so the controlling program must instantiate a value for each each command according to the situation at hand. Except for a few commands that can be sent simultaneously, the simulator only accepts one command per agent per simulation cycle. When a client program sends multiple commands that cannot be executed within a cycle, the simulator randomly chooses one of the commands received from a given player.

The simulator introduces a small amount of random noise to all of the information contained in the visual feedback and some of the information in the self-awareness feedback. It also adds noise to the actions of players and to the motion of the ball after it is accelerated by a kick. As a result of this noise, agents perceive slightly noisy versions of positions of landmarks and other agents. This makes the triangulation of agent positions more difficult. In addition, an agent has a limited view of its surroundings. The Server also adds noise to the commands sent by each player so that the exact effect of each action is uncertain.

2.10.2 Simulator Environment

This section is an abbreviated description of the simulator environment. The soccer environment consists of a maximum of 22 players, the ball, a referee, and landmarkers. Each team has the option to use a coach. The simulator is configured using an input file that contains parameters and their values. The Soccer Server is described in detail in the SoccerServer Manual [Chen et al., 2002].

Soccer Field and Landmarks

The Soccer Server simulates a 2-dimensional soccer field of size 105×68 distance units. The field contains fixed virtual markers to help agents with self-localization and other field-related reasoning as shown in *Figure 2.1*.[2]

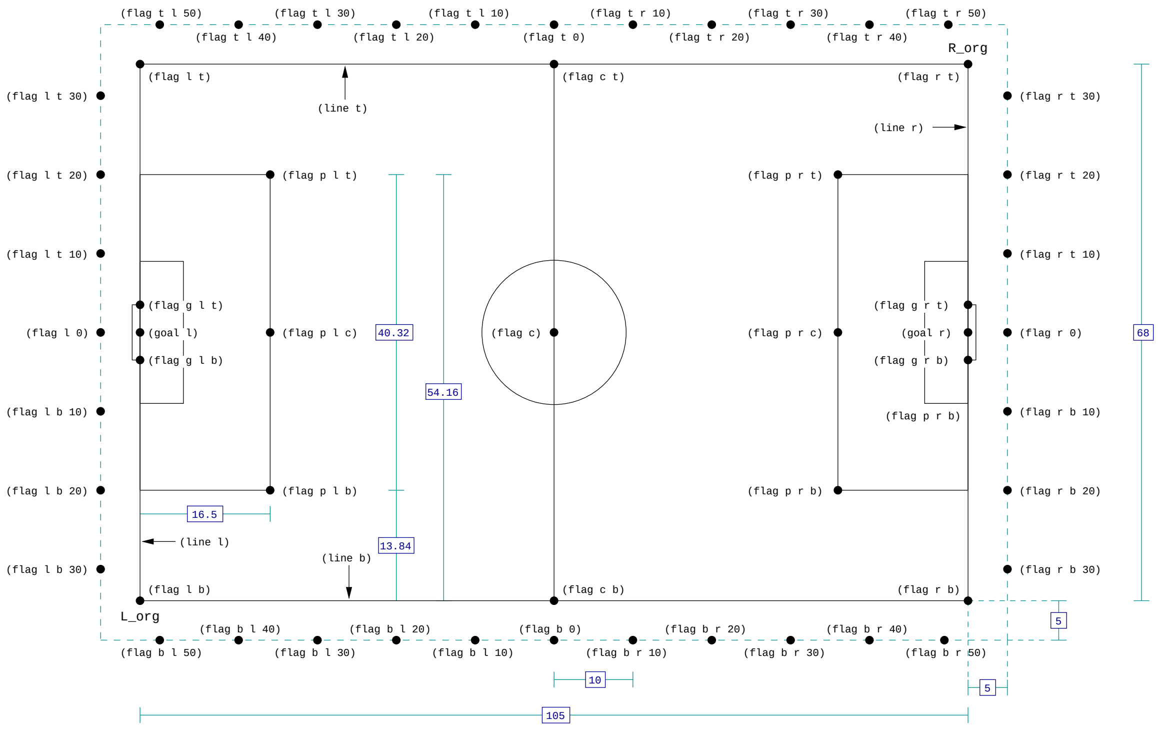


Figure 2.1: The soccer field and fixed reference markers in the simulator environment

Movable Objects: The Ball and the Players

All moving objects are modeled as circles whose radius is specified in the input configuration file. Distances and angles are reported with respect to the center of the circle representing each mobile object.

The environment simulates up to 11 players on each side but not all players need to be used in any given session and the number of players on each side need not be equal. Each player may send a command to the simulator any time during the current simulation

[2]This figure has been reproduced with some additions from the Soccer Server Manual [Chen et al., 2002]

cycle, but all commands sent to the simulator by currently simulated players take effect simultaneously at the end of each simulator cycle.

The direction of an agent's motion is determined by its current body angle, and the direction of the ball's motion is determined by the direction of the kick. In addition, the effect of the movement actions of the ball (following kicking) and the players (following running) last past the cycle in which they are executed. For example, if a player starts running in cycle t, it will continue to move in the direction of its movement in cycles $t+1$, $t+2$ and so on. However, the movement decelerates every cycle, because the simulator decays the movement of each movable object by a factor specified for that type of object in the configuration file.

The simulator also checks for object collisions at the end of each cycle. If any two object circles overlap, the simulator moves the overlapping objects such that they are no longer in collision with each other.

Agent Sensing

An agent's orientation on the field is determined by its position and its body angle. Version 5.00 of the Soccer Server introduced the ability to turn the neck independently but relative to the body so a player can move in a direction different from that of its neck.

The detail and the quality of the information an agent perceives depends on the angle of its neck, its distance from other objects, and the angle of its viewing cone about the direction of its neck. The angle of the viewing cone can be dynamically changed, but the detail of the visual feedback an agent receives is inversely proportional to the width of the viewing angle. The narrower the viewing angle, the more detail an agent receives in its visual feedback about the environment. But, with a narrow viewing angle, the agent does not see as much of its surroundings as it could have with a wider viewing angle. So, for example, it may be best to set up the goalie with a wide viewing angle so that it can see more of the field at any given instance, but, in wide viewing angle mode, there is less information provided about seen movable objects that could otherwise be used to deduce more information about them such as velocity.

A player also receives information about objects that are close-by but not within its current viewing cone. For example, if the ball is behind a player, the player will still receive information about the ball so that it can know the ball is nearby. In the case of nearby players that fall outside the current viewing cone of a player, the simulator sends only partial information. This partial information excludes player identity, so that the player cannot directly identify those players or objects but, it can know their object type.

Agent Communication

In the Soccer Server environment, agents can send and receive messages of a fixed size. All messages are broadcast to all the players within an allowed distance radius. However, each player can only hear a message once every two cycles from its teammates, therefore, sending more messages than what can be heard does not affect the communicative ability of the team.

The referee also makes announcements about the state of the game such for fouls, kick-ins, and offsides. All players, by default, hear all referee messages without limitation. Each player also hears its own messages, but a player's own messages do not impose any hearing limitations as those from teammates would.

Agent Actions

Agents can move in the simulator world based on their *stamina*, which is an exhaustible resource. The Soccer Server decrements the stamina of a player that sends a command to run, but the player can recover part of that stamina every cycle, and especially while it is not moving and hence not expending more of its existing stamina.

The primitive movement-related actions of a player are as follows:

turn(`relative_angle`) : This command turns the body of the player, but the exact amount of the turn is determined by the current speed of the player and is given by the formula,

$$\texttt{actual\_turn\_angle} = \frac{\texttt{relative\_angle}}{1.0 + (\texttt{inertia\_factor} \times \texttt{current\_player\_speed})}$$

where `inertia_factor` is a constant specified in the input configuration file. Therefore, the actual turn angle will be smaller if the player is moving fast than it would be if the player were stationary. The standard range of the *relative_angle* is [-180 . . . +180] degrees.

turn_neck(`relative_angle`) : Unlike the *turn* command, the *turn_neck* command is not affected by the speed of the player. After this command is executed, the head of the player is turned by the indicated amount relative to the body of the player. The standard range of the *relative_angle* is [-90 .. +90] degrees. The *turn_neck* command can also be executed within the same simulation cycle as *dash*, *kick* or *turn*.

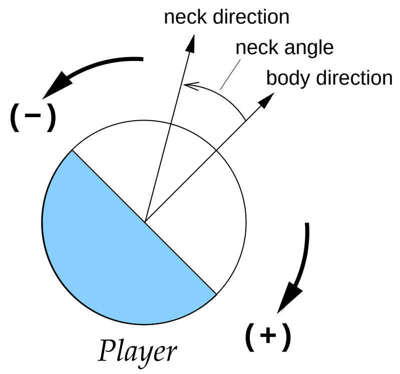


Figure 2.2: Neck angle, body angle, and direction conventions for clockwise and counterclockwise angles

Figure 2.2 shows the direction of the body and that of the neck and the angle conventions used in the simulator. Positive-valued angles refer to clockwise turns, and negative-valued angles refer to counterclockwise turns.

dash(`power`) : The player uses the *dash* command to run in the current direction of its body. As mentioned earlier, the effect of a single running command is decayed after it is executed. Therefore, for sustained running, the player must send *dash* commands in several consecutive cycles. However, it is not possible for a player to keep running forever, since every acceleration obtained by a *dash* consumes stamina. Moving in reverse to the current direction of the body is also possible with a negative *power* value; however, negative power values consume twice as much stamina. The *dash* command is the only command that consumes stamina.

kick(`power, relative_direction`) : The kicking command accelerates the ball in the specified direction that is relative to the direction of the body. The player is allowed to execute a kick if and only if the ball is within a certain distance from the player. Otherwise, the kicking command does not affect the ball.

The strength of the acceleration is determined by a formula (See [Chen et al., 2002]) that takes into account the *power* value and how close the player is to the ball. The closer the player is to the ball, the harder the kick will be.

catch(`relative_direction`) : A goalie can use this command to catch the ball, which must be within an area defined by a standard length and width and the direction argument to the command relative to the current body angle. *Figure 2.3* shows an example where the ball is oriented 60 degrees (counterclockwise) to the left of the current body direction of the player. If the player sends *(catch -60)*, it will catch the ball. The values for the length and the width of the region within which the player can catch the ball are among parameters specified in the input configuration file.

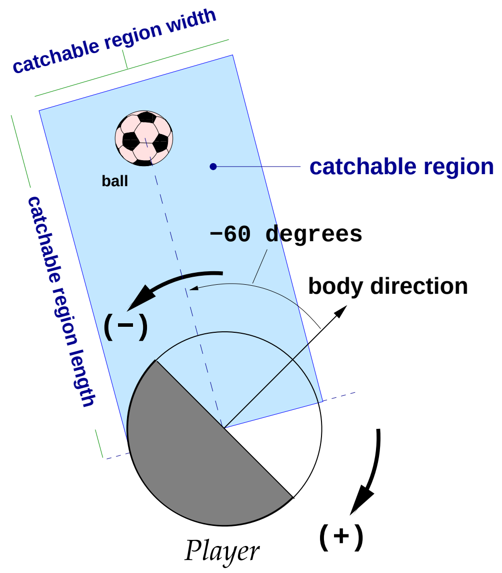
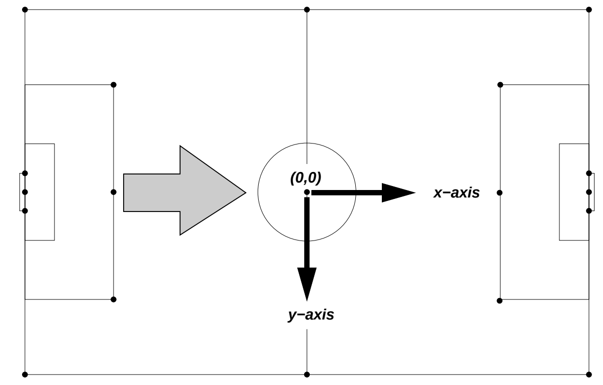
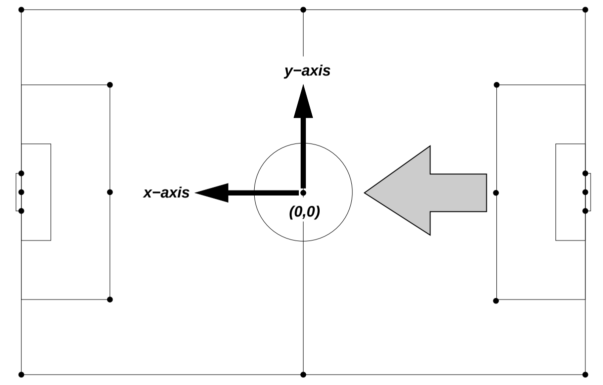


Figure 2.3: An example of ball catching in simulated soccer. The the player can catch the ball if it issues the command, *(catch -60)*

move(x, y) : This command moves a player to position (x,y) on the field. The Soccer Server provides a side-independent coordinate system for each team to place their players on the field at the beginning of a game. This convention is given in *Figure 2.4*. In both cases, the middle of the field is the origin, (0,0), and the positive x-axis extends towards the opponent's goal. The y-axis of a team is at 90-degrees clockwise from its positive x-axis. Due to this relative convention, for the left team, the x-axis extends to the right, and the y-axis extends downwards as shown in *Figure 2.4(a)*. Similarly, the x-axis of the right team extends to the left of the field, and the y-axis extends upward as shown in *Figure 2.4(b)*.



(a) Left field team coordinate convention



(b) Right field team coordinate convention

Figure 2.4: Soccer Server field coordinate conventions for left and right teams. The large arrows indicate the direction of the opponent's half. Note that the coordinate system of the right team is a 180-degree rotated version of that of the left team, or vice versa

The *move* command can be used in three ways: (1) A player can place itself on the field before the game starts, or (2) a goalie can move anywhere within the goalie box after it catches the ball but only a fixed number of times after each successful catch, or (3) a coach can move any movable object to anywhere on the field at any time, before or during a game.

Soccer Server: a Multiagent Testbed

We chose the Soccer Server, the RoboCup soccer simulator environment, as the testbed for our multiagent learning approach. Soccer Server provides a rich domain that is highly dynamic, complex, and uncertain. The game of soccer is also naturally suited to being modeled as a multiagent system. Agents on a given team have to work collaboratively to be successful. In addition, they have to overcome the noise in their actuators (turning, kicking, moving) and sensors (visual perception) as much as possible. They also have to operate well without requiring extensive communication, since the simulator environment places strict limitations on how frequently each agent can communicate and how much bandwidth each message can use. Each agent also has to reason about resources, particularly about its stamina, which is reduced when a player runs. [3] Finally, the highly dynamic nature of soccer requires fast action times. Therefore, deliberation and reactivity have to be balanced. For these reasons, we believe the Soccer Server is very suitable for studying our approach to multiagent plan application learning.

2.11 Summary

Even though all multiagent systems share the same broad set of issues such as architecture, group activity (cooperation, coordination, or collaboration), reasoning, learning, reaction, and communication, the emphasis varies according to the environment for which a system is being built. In this dissertation, our emphasis is on learning in dynamic environments. However, complex domains make it difficult to work on one particular aspect of a

[3] Other actions such as kicking or turning do not consume stamina, and not running causes some stamina to be automatically recovered every cycle.

multiagent system to the exclusion of others that are closely related to the main goal. For this reason precisely, we first presented a historical perspective on where multiagent systems fit in the broad family of AI systems. This chapter also discussed a variety of the aspects of AI systems that closely relate to our aim in this thesis. The next chapter will present the methodology behind this thesis.

◇

Chapter 3

Methodology

This dissertation investigates the problem of *how we can enable a group of goal-directed autonomous agents with shared goals to behave collaboratively and coherently in complex domains that are highly dynamic and unpredictable*. In this chapter, we present our approach to this problem called *mu*ltiagent *r*eactive plan *a*pplication *l*earning (MuRAL). We implemented our approach experimentally in the RoboCup Soccer simulated robotic soccer environment [Kitano et al., 1997b,a; Chen et al., 2002] discussed in *Section 2.10.1*.

This chapter is organized as follows. *Section 3.1* describes the motivation for symbolic multiagent learning in complex, dynamic, and uncertain domains. *Section 3.2* is an overview of the MuRAL approach. *Section 3.3* is a detailed description of MuRAL. *Section 3.4* discusses the primary algorithms we use in our work. *Section 3.3* and *Section 3.4* together form the bulk of this chapter. *Section 3.5* describes the method we use to evaluate the performance our the approach. *Section 3.6* describes the experimental setup we used for learning and testing. *Section 3.7* describes the plans we used to demonstrate our learning approach in this research. *Section 3.8* is a summary of work closely related to our approach. Finally, *Section 3.9* summarizes the ideas presented in this chapter.

3.1 Motivation for Learning

A set of agents and the environment in which they are situated form a *complex system*. A *complex system* is one whose global behavior cannot easily be described by a subset of its true features. This difficulty of description is mainly due to the non-determinism, lack of complete functional decomposability, distribution, and behavioral emergence of such systems [Pavard and Dugdale, 2000]. Because of these difficulties, the dynamic nature of the interactions of agents with the environment and the interdependent interactions of agents with each other cannot be described fully in complex domains. Instead, designers of AI systems manually choose the features that are considered most essential to modeling the overall behavior of that system, and then they base solutions on those chosen features.

The fundamental problem of complex systems is the exponential growth of the state and action search spaces in the number of agents. This means that solution methods that work with a small number of agents may not scale up to larger-sized problems with more agents. Furthermore, if learning is strongly dependent on the action patterns of other agents, concepts that need to be learned may continually shift as other agents themselves learn, and, consequently, learning may not stabilize.

Since real-life complex systems are dynamic and unpredictable, problem solving methodologies to be employed in such systems need to be *adaptive*. A limited level of adaptability can be provided by hand-coding. However, it is difficult to account for all the implementation-level details of a complex system without an understanding of the interdependencies among agents. The nature of these interdependencies is unfortunately highly context-dependent. Therefore, a situated learning approach is better suited for capturing the critical features of the domain to create adaptable systems than a non-learning approach.

Because we are dealing with complex environments where agents pursue long-term goals, the learning approach has to implement a balance between *reaction* and *deliberation*. Commitment to long term goals is essential for a multiagent system [Bratman et al., 1988], and, at the same time, agents need to be able to react to dynamic events in order to be able to continue to pursue their long-term goals.

In a complex environment, an agent may be able to make some limited predictions about the state of the world within a very short window of time. However, an agent views its environment only from a local perspective. So, even if it is theoretically possible to construct a more global view of the environment by exchanging local views among several agents, it may take relatively a long time to construct such a global view. During this time, the environment may change in significant ways to render that global view only partially correct or relevant. If we take into account that factors such as communication bandwidth, opportunities for message exchange among agents, and communication reliability may be severely limited in a real environment, we can conclude that the adaptability effort cannot afford to require that each agent have a global view of its environment. Even if a global view can be built, it will still be unclear whether having to deal with the resulting larger state space would be helpful or prohibitive to adaptive behavior.

An added difficulty in real-world systems is the noise in the sensors and actuators of each agent and the uncertainty in the environment largely due to external factors that are difficult to predict. An agent's sensors provide only a local view of the environment, and, moreover, the information they provide is imperfect. Actuators do not necessarily execute each command flawlessly to bring about the intended changes without any margin of error. Conditions external to an agent may also affect the outcome of each executed actuator command.

It is possible that a multiagent system with a static library of situation-action rules can exhibit collaborative goal-directed behavior. However, such a system would have only limited adaptability, and it would be unable to improve its behavior over time. Such a system would require a large amount of *a priori* knowledge of all states it can be in, or it would have to contain enough number of generalizations so as not to be ineffective. Similarly, a traditional planning approach would also be insufficient, since planning approaches are not adaptive to highly dynamic environments. An interleaved planning and reaction approach could conceivably provide a solution. However, it would also have limited adaptability. One prevalent problem in all these non-learning approaches is the need for a method to capture the fine details of each specific situation with both its dynamic context description and the associated action sequences from each involved agent's perspective. Therefore, we believe a learning approach that enables each agent to learn context descriptions and effective action sequences in a situated fashion can provide a better solution to multiagent problem solving in complex, dynamic, and uncertain environments than non-learning approaches can.

Thus, the learning methodology for a goal-directed collaborative multiagent system that needs to operate in a complex environment has to address the critical challenges posed by large search spaces, noise and uncertainty, and concept shifts to achieve adaptability in realtime.

3.2 Overview

In complex, dynamic, and uncertain domains, agents need to be adaptive to their environment to accomplish their intended tasks. When collaborating in a team, each agent has to decide on what action to take next in service of the relatively long-term goals of its team while reacting to dynamic events in the short-term. Adaptability in dynamic and uncertain environments requires that each agent have knowledge about which strategies to employ at the high level and have knowledge about action sequences that can implement each chosen strategy at the low level in different situations and despite potentially adverse external conditions. Moreover, agent actions must be coordinated such that collaboration is coherent. The knowledge required for these tasks is unfortunately heavily context-dependent and very variant, and, therefore, it is difficult to obtain manually. With learning, on the other hand, agents can have the continuous ability to automatically acquire the necessary knowledge through their experiences in situated scenarios.

In this dissertation, we target complex, dynamic, and uncertain domains where agents need to work in teams towards common goals. Such environments can change very quickly, and therefore, they require fast or realtime response. We assume that agents can sense and act asynchronously. Each agent operates and learns autonomously based on its local perspective of the world. We assume that the sensors and the actuators of agents are noisy. Agents cannot obtain precise information about their environment using their sensors, and they cannot affect their environment in exactly the intended ways at all times. Since the environment has uncertainties, the results of agent actions cannot be reliably predicted. Therefore, building a model of the environment and the behavior of other agents (whether these are agents that the current agent needs to collaborate with or compete against) is an extremely challenging task. Moreover, we assume that agents do not communicate due to bandwidth limitations and communication channel noise. Yet we require that agents manage to work collaboratively with each other in a team and learn to perform tasks in situated scenarios with possibly other agent groups with different or adversarial goals operating within the same environment.

In recent years, Reinforcement Learning (RL) techniques have become very popular in multiagent systems research. The RL approach is suited for sequential decision making since it allows agents to learn policies based on the feedback they receive from the environment in the form of a reward signal following each action execution. As an agent executes its actions, RL allows it to adapt its behavior over time via the reward signal. Over many training iterations, each agent learns a policy, which maps the environment states to its actions. However, this standard RL formulation does not directly apply to complex real-world domains. Therefore, the standard formulation has been modified by some researchers to make RL applicable to realistic multiagent systems [Stone and Veloso, 1999; Matarić, 1991, 1996]. Despite the modifications, these approaches still suffer from a set of problems that make them difficult to learn complex policies and scale up to problems with many agents.

Bottom-up techniques such as RL require convergence for the stability of the learned policy. Convergence means that these techniques learn only one policy [Matarić, 1991]. In complex domains, however, it is very difficult for such a policy to cover all aspects of agent behavior. It is rather the case that a very restricted set of tasks can be learned using such techniques for a relatively small group of agents. These problems emanate mainly from having to deal with large search spaces.

In multiagent learning, the state space grows exponentially with the number of agents introduced into the environment. Since a RL method has to search the entire search space, the exponential growth of search spaces is detrimental to convergence even when search is limited to the local view of each agent. Therefore scaling up solutions to larger problems is very difficult. One other related problem is that emergent learning techniques have to handle concept shifts. As an agent learns, the policy it has learned so far has to be continually adjusted to account for the changes in the behavior of other agents due to their own learning. This is especially the case in adversarial environments [Stone, 1998]. Also, in RL, the policy to be learned is only implicitly defined by the reward function. Thus the search for a policy or a behavior is much less constrained than it is in our focused search approach, and this lack of top-down control makes these approaches more difficult to scale up.

The goal of *multiagent reactive plan application learning* (MuRAL) is to provide a methodology for the automated implementation of symbolically described collaborative strategies in complex, dynamic, and uncertain environments. We assume that strategies are symbolically described as *plans* at an implementation-independent, high level by human users who are possibly experts in a domain [Sevay and Tsatsoulis, 2002; Myers, 1996, 1997]. A plan is divided into distinct steps, and the task in each step is divided among a set of *roles*. Each *role* initially represents only the necessary conditions for executing and terminating the specified responsibilities of an agent contributing to the implementation of a plan step. Therefore, collaboration is implicit in the definition of plans. A plan, does not, however, contain any implementation-specific knowledge at first. The purpose of learning is to acquire this knowledge from situated experiences to enable the execution of high-level strategies in various situations.

What is new in our approach is that we use symbolic high-level multiagent plans to constrain the search from top-down as agents learn from bottom-up. The MuRAL approach deals with large search spaces by constraining the learning task of each agent to the context of each explicitly defined reactive multiagent strategy such that each agent knows what specific aspects of the environment it needs to account for in order to learn how to implement its responsibility for each plan step. Since each plan is divided into distinct steps for each role, learning is further compartmentalized to the perspective of each agent that assumes a given role in a plan. Hence, search constraining at the top level and learning at the low level complement each other and help reduce the search space. Using explicit plans makes it possible to concentrate the learning effort of a group agents on filling in the dynamic details of their plans to implement given strategies, rather than discovering entire strategies in a bottom-up fashion as in bottom-up learning techniques such as Reinforcement Learning (RL).

As it is in our approach, situated learning is also fundamental to RL systems, except that, in the MuRAL approach, agents do not learn policies. Rather, they use learning to acquire implementation details about given plans. Agents acquire action knowledge in order to *operationalize* plans and collect *reinforcement* information regarding their plan execution to be able to choose the most useful plans to apply in future situations, and, within each chosen plan, to pick the most appropriate implementation for each plan step.

Learning in our approach is divided into two phases. In the *training phase*, agents acquire knowledge to operationalize each plan step in various situations. These training scenarios are situated but controlled, and agents use no communication. To acquire operationalization knowledge for each plan step, each agent uses best-first search and case-based learning (CBL). Best-first search is used to find alternative sequences of high-level actions that solve the goal of a plan step in a certain situation. A description of each situation and an action sequence that solves the problem in that situation are stored as a case in the local casebase associated with the corresponding plan step. The operators used in the search correspond to high-level reactive behavior modules rather than primitive actions. Besides reducing the search space, the use of high-level behavior modules enables agents to reactively respond to dynamic changes in the environment.

In the *application phase*, each agent selects a plan and attempts to execute it to completion in full-fledged scenarios. As in the training phase, agents use no communication. While executing a plan, each agent collects reinforcement depending on the success or the failure of each plan step as well as the entire plan. To apply a given plan step to the current situation, an agent retrieves all similar cases from its local casebase of the current plan step and selects one non-deterministically. The probability of picking a given case is a function of the success rate of that case, and cases with higher success rates have higher probability of being selected for execution.

An agent deals with the uncertainty in its environment by reinforcing plans and plan step implementations from its own experience such that successful implementations get favored more than the unsuccessful ones over time. If an agent applies a plan step successfully, then that plan step gets reinforced positively for the corresponding role. If the application fails, the plan step receives negative reinforcement. When all steps in a plan are successfully applied, then the entire plan is reinforced positively. In case all plan steps cannot be applied successfully to the current situation, then the plan gets negatively reinforced and the plan steps until the failure point keep their reinforcement assignments. Since agents do not communicate during the application phase, coherence among agents during collaboration depends on this reinforcement scheme.

Communication among many agents to establish perfect synchronization in a complex, dynamic, and uncertain environment is very costly and likely prohibitive to coherent behavior. Agents would have to exchange multiple messages before deciding which plan to execute and how to distribute each multiagent task among themselves. Since different sets of agents can potentially execute a plan in a given instant, choosing the exact group of agents would require some form of negotiation. However, the number of messages that would need to be exchanged among agents before a final decision can be made would disable reactive behavior in dynamic environments and severely limit scalability. Therefore, our approach does away with any message exchange, and, instead, each agent evaluates the progress of each plan to the extent of its sensing and reasoning capabilities.

3.3 Multiagent Reactive Plan Application Learning (MuRAL)

The MuRAL methodology makes the assumption that a user can provide high-level plans to guide the solution process. A plan describes only the necessary parts of the implementation of a high-level strategy that may require the collaboration of multiple agents. A plan is divided into distinct consecutive steps, and each plan step specifies a set of *roles* that need to be filled by the agents that commit to executing that plan. A *role* describes the necessary conditions for execution and termination of a given plan step and the responsibilities that an agent contributing to that plan step must take on.

Each plan step specifies *what* needs to be operationalized from each role's perspective but does not specify *how* to do so. Therefore plans are designed to guide the system behavior rather than dictate all runtime details, which can potentially have infinite variation in a complex domain. It is then the job of the multiagent system to automatically learn, through training in a situated environment, the fine details of how to operationalize each plan step in varying contexts.

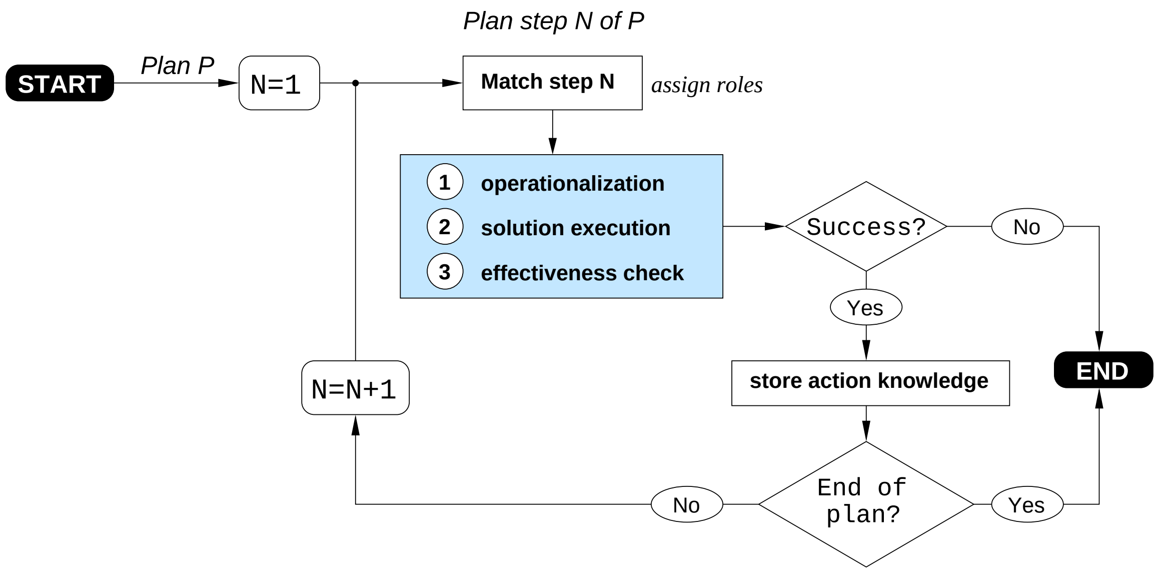


Figure 3.1: The training phase

A MuRAL system operates and learns in two phases. The first phase is the *training phase*, where, given a plan, each agent acquires action knowledge to operationalize its responsibilities described in that plan by the role it assumed. The second phase is the *application phase*. In this phase, each agent executes a given plan that it has already learned to operationalize during training. The agent learns which steps in that plan are more effective than others using a naïve form of reinforcement learning. *F*igures 3.1 and 3.2 depict the flow of these two phases.

The training phase shown in *Figure 3.1* is made up of two parts that are interleaved. In the *operationalization phase*, the goal of the multiagent system is to operationalize each

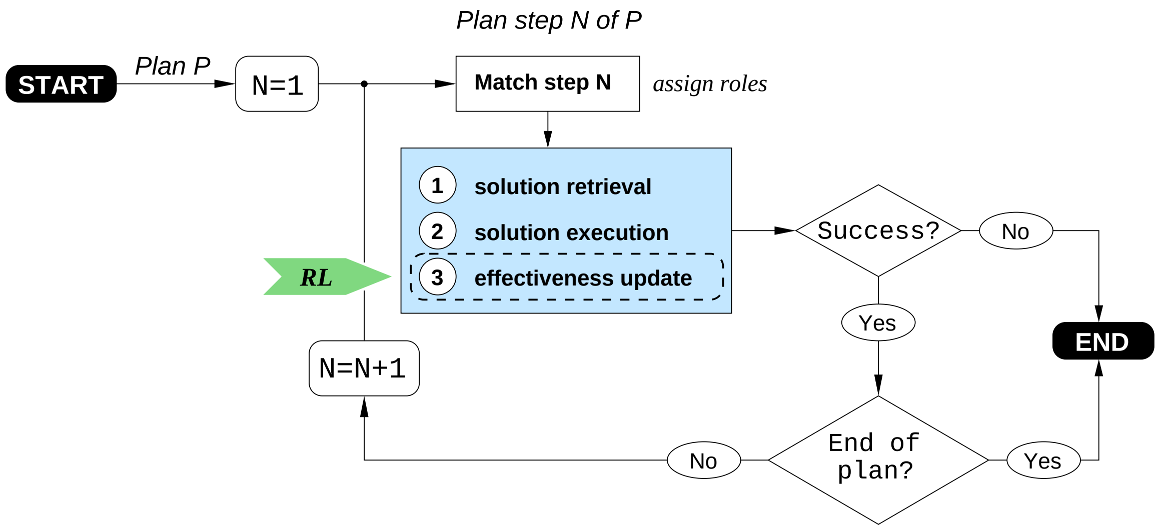


Figure 3.2: The application phase

step of a given plan from each agent's perspective, thereby operationalizing the entire plan after all steps are completed (marked as ① in *Figure 3.1*). In the *feedback phase*, the agent executes the solutions of the reasoning phase in the actual environment and checks both whether each plan subpart has succeeded to implement the desired subgoal and whether the entire plan has succeeded or not (marked as ② and ③ in *Figure 3.1*). The agent stores the working solutions in each plan step so that those solutions are available for the application phase if training succeeds for the entire plan. A training episode ends either when the entire plan has been successfully applied or a failure occurs. At each plan step, the agent executing a given plan assumes a fitting role for itself and assigns roles to other agents it is collaborating with. These role assignments are internal to each agent and are not communicated among agents.

The application phase shown in *Figure 3.2* also comprises two interleaved parts. The *solution retrieval phase* in application involves matching the current situation to the knowledge content of an agent so that solutions acquired during training can be used to solve similar problems (marked as ① in *Figure 3.2*). The *feedback phase* of application is similar to that of training. In application, an agent may access a given piece of action knowledge many times, thereby modifying its effectiveness status over time (marked as

② and ③ in *Figure 3.2*). In addition, the application phase tracks the effectiveness of each executed plan. A plan may fail at any time, but it will only succeed if all steps in that plan have been successfully completed. This feedback collected over a series of situated tests allows agents to select more effective plans and plan implementations. Therefore, the system learns in both the training phase as well as the application phase.

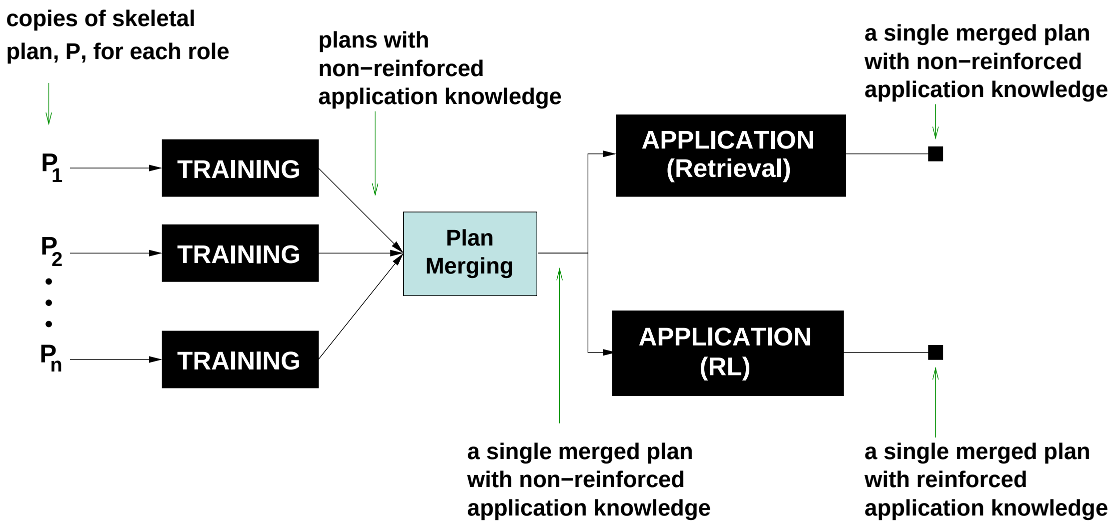

copies of skeletal plan, P, for each role
plans with non−reinforced application knowledge
a single merged plan with non−reinforced application knowledge
P1
P2
Pn
TRAINING
TRAINING
TRAINING
Plan Merging
APPLICATION (Retrieval)
APPLICATION (RL)
a single merged plan with non−reinforced application knowledge
a single merged plan with reinforced application knowledge

Figure 3.3: Summary of the MuRAL methodology

Figure 3.3 summaries the MuRAL methodology. The system starts with identical copies of a given skeletal plan, P. Each agent collaborating to execute P has its own copy of P for practical purposes. After each successful training trial, each copy of P, P_1, P_2, $..$, P_n, will contain potentially unique knowledge for executing P in a specific situation. Since this knowledge has not yet been used during learning, it is non-reinforced. After merging the knowledge from all successful trials for P, we obtain a single *operationalized* plan that contains all the knowledge for all roles. This single plan can then be used for retrieval or learning. Since retrieval does not modify plans, each given plan stays unchanged at the end of retrieval. RL, on the other hand, modifies the operationalized plan it started with.

3.3.1 Case-Based Learning and Naive Reinforcement Learning

Since dynamic environments continually change, an agent may be able to predict future events only within a very short time window, and the number of possibilities to consider during the next action decision is very large in a complex and highly dynamic domain. Therefore, agents cannot afford planning a relatively long sequence of actions to execute in the very near future. Generating plans in the traditional planning sense is also impractical since an agent cannot predict all events in the future and cannot afford to plan for every possibility. An interleaved planning and execution approach [Ambros-lngerson and Steel, 1988; Wilkins, 1984] is also not suitable, because an action sequence discovered during search cannot be guaranteed to work in the real environment. However, agents have to be able to make progress towards their goals by keeping their commitments to the plans they have selected to execute as much as possible [Bratman et al., 1988]. Since agents need to operate in a dynamic environment, each agent can afford to plan relatively short sequences of actions at each decision point, unless the environment happens to change at a rate that renders any reasoning useless. Then the agent can test these short plans in the real environment to find out if they work under the conditions they were generated for. The agent can retain the working action sequences and the corresponding situation descriptions in its memory. Without feedback from the environment, however, it is impractical to expect an action sequence generated during planning to work in actual scenarios. This is the reason why the MuRAL approach relies on the feedback that agents draw from the environment in the form of reinforcements.

In order for reasoning to take place, any given domain must be assumed static at some time scale and for some duration. That is, we have to make a *temporally static world assumption*. This assumption only facilitates reasoning, but it does not guarantee that the action sequence an agent discovers will necessarily serve its intentions. The reason for this lack of guarantee is the complexity and the uncertainty of the domain. The agent cannot predict the future states of its local environment entirely even if it can make some predictions about itself at each decision point.

An agent can make the assumption that all other agents, including its group members and others, will behave in the same manner in similar circumstances. However, in order to make predictions about the actions of other agents, the agent has to take into account the interdependencies among the agents, since agents in a multiagent system frequently base their decisions on the behaviors of others. Reasoning about interdependencies is, however, an infinite regression problem, and, even at limited depth, it would be too costly to employ in dynamic environments.

For efficient behavior in a complex system, each agent needs to have three types of knowledge through the following processes:

1. *Action Sequencing*: Suppose that we are given that plan P expresses some strategy G. For each subgoal g of G, the agent needs to a determine a sequence of actions, s, that will solve g under the *temporally static world assumption* in some situation c. That is, s is an *operationalization* of g under c.

2. *Subgoal Application Feedback*: The agent needs to know that a given action sequence s can efficiently solve subgoal g in the actual environment under c. If there are multiple action sequences that are applicable to g under c, then the agent can pick one from this set based on the success rate of each alternative s.

3. *Strategy Application Feedback*: The agent needs to know that applying P in similar situations will efficiently implement G. That is, P is an efficient implementation of G, given the action sequences and feedback retained for its subgoals.

Action sequencing requires reasoning and uses the world model that each agent keeps in its memory, and it takes place during the *training phase* of an agent's operation. Feedback about subgoal and strategy application, on the other hand, has to do with first-hand interaction with the environment since agents execute actions in situated fashion, possibly interacting with other agents in the process. Both feedback processes take place during the *application phase*.

Since the entire dynamics of a complex system cannot be expressed in a formal model, it is critical for each agent to test the action sequences it generates during reasoning in the actual environment. In this way, the dynamics of the environment that is relevant to the current subgoal g can be captured by the test and summarized by whether the action sequence in question has solved g or not. Without any feedback from the environment, the agent cannot know if a given action sequence s can potentially solve the current subgoal g or a certain plan P can implement strategy G under the current circumstances. So, for example, if the agent generates n different alternatives in the action sequencing step, it will not know which to pick, since all alternatives will seem equally valuable. Therefore, feedback based on direct application of action sequences generated during reasoning is required to distinguish among potential alternatives.

In action sequencing, each agent retains action knowledge about a given subgoal, and, in the second step, it retains feedback about the application of its action knowledge from the first step. This means that, in order to apply a plan in the future, the agent has to be able to identify which action sequences are appropriate for the current situation. In short, both steps involve some form of learning.

Assuming a temporally static world, we formulate the action sequencing step as a search problem [Korf, 1987]. We use best-first search [Rich and Knight, 1991] with domain-specific heuristics to implement this approach. The search operators we use in the best-first search represent high-level reactive behaviors rather than primitive actions. We call these high-level reactive behaviors *reactive action modules* (RAMs) and discuss them further in *Section 3.3.5*. For example, in the simulated soccer domain, passing the ball to another player usually requires a number of calculations and a sequence of different primitive actions whose exact makeup and parameter values are highly dependent on the dynamics of the current situation. Therefore, using high-level representations of behaviors as search operators as opposed to primitive actions reduces the search space considerably. Especially in multiagent dynamic environments, it is unrealistic to plan at the level of primitive actions, since runtime conditions will require reactions that an agent could not possibly plan for in advance. The search heuristics we use bias the search towards finding the shortest possible sequences of actions. However, that an action sequence is the shortest possible that can be found is not by itself a guarantee for success in the actual environment. Hence the need for direct feedback from the environment.

We formulate this step as a *naïve reinforcement learning* problem, where the statistics of past successful or failed applications of action sequences are used to distinguish among alternatives. This approach is reinforcement based, but is different from what-is-known-as RL [Sutton and Barto, 1998]. The purpose of naïve reinforcement learning is to enable agents to choose plan step implementations from among the most useful ones in past similar situations. In our approach, this selection is based on the success rate of plan step implementations. Generally speaking, an agent positively reinforces a plan step if it succeeded in the current situation and negatively reinforces it if it failed. The dynamics of a complex and uncertain environment, however, require more refined reasoning so that agents can achieve a level of coherence possible in such domains.

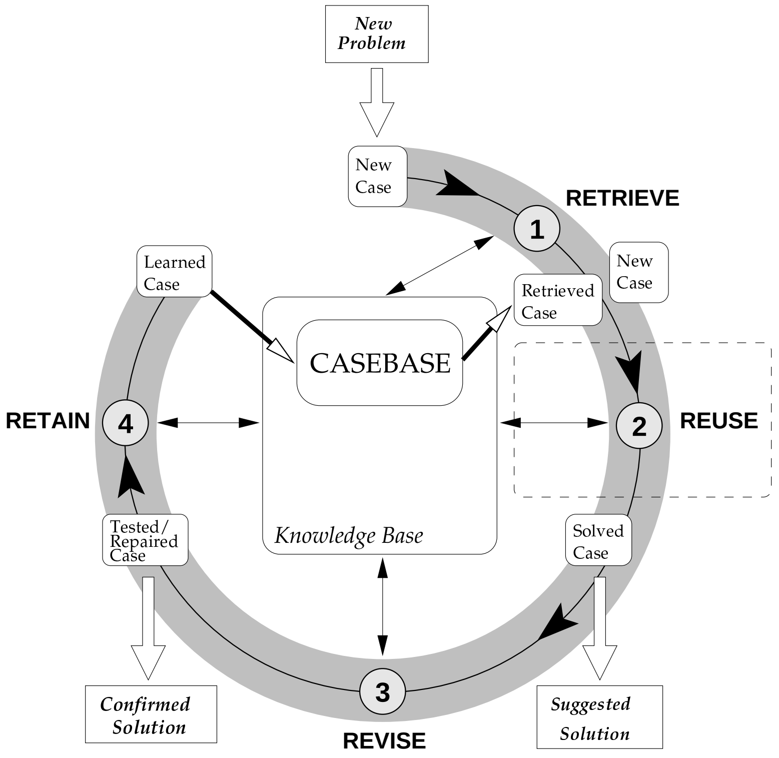


Figure 3.4: Case-based reasoning cycle

MuRAL uses case-based reasoning [Tsatsoulis and Williams, 2000] to select plans by matching plans to situations (See *Section 3.3.6*). It uses case-based learning [Aha, 1991; Aamodt and Plaza, 1994] to match plans to situations at the high-level and match action sequences to implement each plan step at the low-level. According to [Aamodt and Plaza, 1994], the case-based reasoning (CBR) cycle is composed of four steps, as depicted in *Figure 3.4*: [1]

1. *Retrieve:* Retrieve the most similar case(s) from the casebase.
2. *Reuse:* Use the information and knowledge in retrieved case(s) to adapt a solution for the current problem. The solutions at this stage are only suggestions, since they have not been yet tested.
3. *Revise:* Evaluate the new proposed solution by applying it in the real environment and possibly repair the solution to make it fit to the current situation. After this step is completed, an agent has a confirmed solution, assuming the suggested solution from the *Reuse* step has been successfully applied to the current situation.
4. *Retain:* Store the useful parts of the new solution that are likely to be reused in future problems in the casebase either by modifying an existing or introducing a new learned case.

A new problem is introduced into the CBR cycle as a new case. After a similar case is retrieved from the casebase (*Retrieval*), the solution in that retrieved case is a suggested solution for the new problem. This suggested solution is then applied to the current situation in the *Revise* step. The purpose of CBL in MuRAL is to enable approximate situation matching and hence deal with the search space complexity. In a complex and dynamic environment, all aspects of physical situations cannot be defined in exact terms. For example, the position of an agent in the world cannot be expected to be identical in every similar situation (*Section 3.3.6*).

In MuRAL, following this testing in the real environment, a retrieved case, in general, receives positive or negative reinforcement depending on whether its execution was successful or not. Then, in the *Retain* step, this modified case is stored back in the case base. Besides using the casebase, an agent may also use other general-purpose or domain-specific knowledge at any stage during the CBR cycle. It is important to note that the CBR

[1]The figure was adapted from [Aamodt and Plaza, 1994] with minor modifications.

cycle does not apply to the *training phase* as in regular CBL, since new cases are learned only through training by storing them directly in the casebase [Aha, 1991]. Thus, the CBR cycle in *Figure 3.4* applies only to the *application phase* of an agent's operation.

The action sequence in a case is determined by a best-first search [Rich and Knight, 1991] (see) and how this sequence is used during the execution of the corresponding plan step is critical to the operation of a MuRAL agent. First, this action sequence is stored in a *case* in summarized form. Let us consider the example in *Figure 3.5* where agent A is in grid square $(4, 1)$. Let us suppose that the solution to a problem A currently faces involves agent A moving left by three grid squares, moving up by two grid squares, moving left by one grid square, and finally moving up by one grid square. That is, this solution sequence moves agent A from grid square $(4, 1)$ to $(0, 4)$. The exact action sequence $S = \{\text{MOVE\_TO}(3, 1), \text{MOVE\_TO}(2, 1), \text{MOVE\_TO}(1, 1), \text{MOVE\_TO}(1, 2), \text{MOVE\_TO}(1, 3), \text{MOVE\_TO}(0, 3), \text{MOVE\_TO}(0, 4)\}$ will have been determined by search.

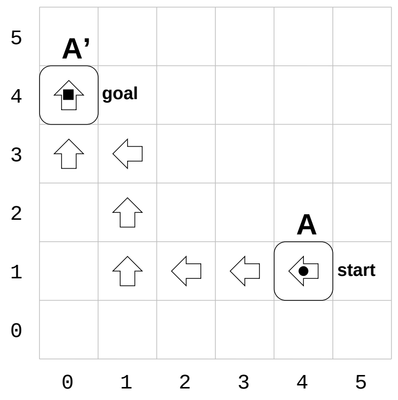


Figure 3.5: Action representation in cases using a grid world

Even though the sequence of actions in S represents the theoretical solution to the problem A faces, it is inefficient to store all seven moves in the ***actions*** slot of a case. Second, and, more importantly, it is inefficient to force an agent to execute the exact action sequence discovered by search, since doing so ties the agent down to too specific an action set to successfully execute in a situated environment. Therefore, instead of storing all seven moves, the agent stores a summary of them. In the case of the example, the summarized action sequence would be $S' = \{\text{MOVE\_TO}(0, 4)\}$.

In a summarized sequence, any static decisions taken during search are dropped such that the essential actions remain. The purpose of doing this is so that the agent, at execution time, is given the most abstract goal specification sufficient to implement the goal expressed in a given action sequence, thereby allowing the agent with as much runtime flexibility as possible. That the search had discovered a sequence 7 actions long to solve the problem does not mean that the same exact action sequence needs to be executed in the actual environment for the successful implementation of the postcondition of the current plan step. Actually, following the exact sequence discovered during search constrains the agent and may even cause failure rather than success in a situated environment due to being overly specific. Therefore, the idea is to keep only what is essential of the results of search so that the runtime flexibility of the agent is maximized. Since the agent will have to make dynamic judgments about how to go from grid square $(4, 1)$ to $(0, 4)$, it may have to avoid obstacles on its way. So it is not even clear that agent A would even be able to traverse the exact path returned by search.

We have to remember that the world description given as input to the search algorithm is assumed to be temporally static. Moreover, the search operators need not implement the details of the corresponding actuator commands that will be used by the agent to implement action sequences in cases. In fact, this would not have been practical also, because of the dynamic decisions that agents need to make depending on the context of the situations they face every moment. A real robot, for example, would have to execute turns in order to travel to the location suggested in *Figure 3.5*, and it would likely have to adjust its speed during the straightaway portions of its planned navigation path. However, none of these considerations need be part of search, since they will have to be dynamically decided on.

3.3.2 Best-First Search

We use best-first-search to determine the sequence of actions to apply to a situation during the training phase [Rich and Knight, 1991]. The search is done in a discretized version of the world where locations are expressed in terms of grid squares rather than exact (x,y) coordinates. We use the following search operators and heuristics to compute an estimation, $f' = g + h'$, of how close a given search node is to a goal. The g value is always the depth of a search node in the search tree:

- Operators $DribbleBall$, $GotoArea$, and $GotoPosition$ are dependent on a destination position or area. When we consider these operators, we use the following heuristic for computing the h' value.

  First, we compute a *proximity heuristic* for each player in the influence area of a rotation:

  1. We compute the Manhattan distance, m, of a player player, p_i, to the current player, P, in the grid world:

     $$m = manhattanDistance(P, p_i)$$

  2. We compute the "adjacency" of p_i using:

     $$adjacency_{p_i} = m + adjacency\_offset$$

     where $adjacency\_offset$ is a small constant set to 0.1.

  3. We compute the "cost" of p_i using:

     $$cost_{p_i} = \frac{player\_cost}{adjacency_{p_i}^2}$$

     where $player\_cost$ is a standard constant cost of a regular player set to 2.0. If p_i happens to be an opponent, then the cost is multiplied by a penalty term, $cost\_penalty$, whose value is set to 5.0 in our implementation:

     $$cost_{p_i} = cost_{p_i} * cost\_penalty$$

  4. Finally, the summation of all player costs gives us the value of the proximity heuristic:

     $$h1 = \sum_{i=1}^{n} cost_{p_i}$$

After computing the proximity heuristic for each player in the influence region of the current scenario, we determine the cost of a search node as follows:

1. First, we compute the distance, d, from P to the target point or area (using the center point as target) using:

$$d = distance(P, target\_point)$$

2. Second, we compute a distance heuristic, $h2$, using the formula:

$$h2 = d^2$$

3. Third, if the operator being considered will undo the effects of the previous operator, then we assign a non-zero penalty cost to the operator:

$$undoCost = 1000$$

Finally, the h' value is computed as:

$$h' = h1 + h2 + undoCost$$

- Operators *InterceptBall*, and *PassBall* are independent of any target coordinates and are always the last action targeted by search. Therefore, their h' value is set to 0.

3.3.3 Agent Architecture

An agent in a MuRAL system operates autonomously and makes decisions based only on its local view of the environment. The architecture of each MuRAL agent is hybrid, with both deliberative and reactive components. *Figure 3.6* shows the architecture of a MuRAL agent where arrows indicate the interaction between components and the agent's interactions with the environment.

Sensors collect information about the environment, and the *Actuators* execute the actions that the *Execution Subsystem* determines using instantiated plans from the *Plan*

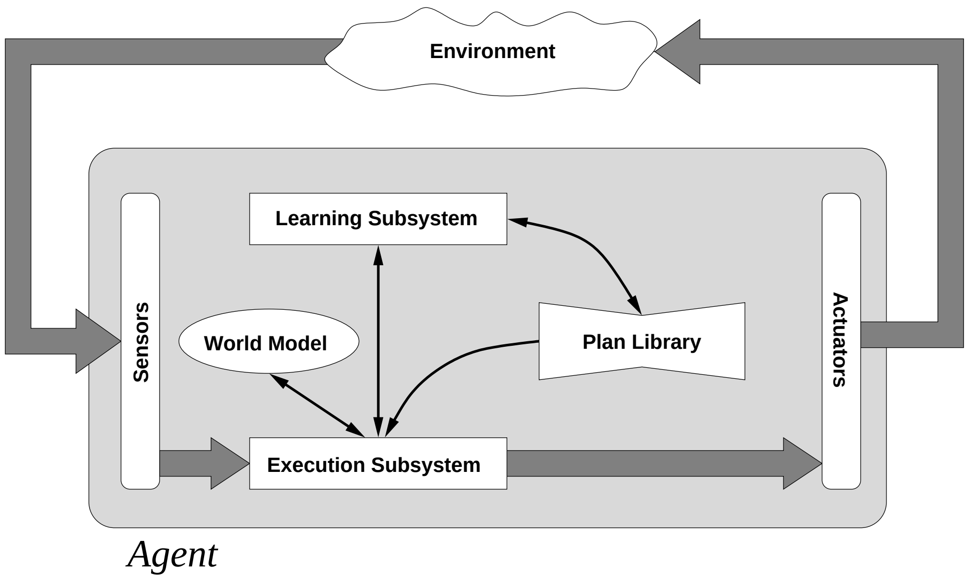


Figure 3.6: The hybrid agent architecture in MuRAL

Library. The *World Model* represents the agent's knowledge about the world. It contains both facts available via the sensors as well as those derived by reasoning about the sensory information. The information that the *World Model* stores is temporal. Over time, through sensing, the agent updates its information about the world, and, to overcome as much as possible the problem of hidden state, it projects facts from known information. Each piece of information in the world model is associated with a *confidence value*. A newly updated piece of information has a confidence of 1.0, meaning completely dependable. The agent decays the confidence value of each piece of information that does not get updated at each operation iteration, and below a fixed confidence threshold, a piece of information is considered unreliable, and it is forcibly forgotten so that it does not mislead the reasoning of the agent. This idea and similar fact decaying ideas have been used in recent systems (e.g., [Noda and Stone, 2001; Westendorp et al., 1998; Kostiadis and Hu, 1999; Reis and Lau, 2001]).

The *Plan Library* contains all the plans that are available to a given agent. Each agent has its own unique plan library. Plans in different plan libraries may contain identical skeletal plans at first, but, after the training and application phases, the application knowledge and the reinforcements of cases in each plan will depend only on the experience of each agent in its situated environment.

Both the *Learning Subsystem* and the *Execution Subsystem* interact with the *Plan Library*. The *Execution Subsystem* accesses both the *World Model* and the *Plan Library*, but it can only modify the *World Model*. The *Learning System* both accesses and modifies the *Plan Library*, and it indirectly uses the *World Model* via the requests it makes to the *Execution Subsystem*. The *Execution Subsystem* then modifies the state of the *Learning Subsystem* in order to affect the *Plan Library*. The *Execution Subsystem* receives sensory information from the environment and updates the *World Model* of the agent. It is also responsible for executing actions in the environment via the *Actuators*.

As *Figure 3.6* indicates, the *Execution Subsystem* is the main driving component of the MuRAL agent architecture. It interacts with the world via *Actuators* and *Sensors*. Its purpose is to select and execute plans in the environment during the lifetime of the agent both during the training and the application mode of an agent. Since it interacts with the environment, it deals with dynamic changes as it executes plans.

3.3.4 Knowledge Representation of Plans and Cases

In MuRAL, plans and cases are represented symbolically. Cases are a natural subpart of plans, but, since they constitute a critical part of plan representation, this section discusses the two data structures separately. *Figure 3.7* shows the top-level representation of a plan.

Plan Representation

A plan, P, is represented by a tuple $\langle N, S, F, B\rangle$. N is the name of the plan, assumed to be unique in a given plan library of an agent. S and F are counters that represent the positive and negative reinforcement for P through the individual experience of a given agent that applied P. Since their is no communication among agents, each agent updates S and F independently based on the conditions in P and its perspective on the world. S stores the number of times P was successfully used, and F stores the number of times P's application failed. Finally, B is the body of the plan.

The body of a plan is made up of a number of consecutive steps that have to be executed in order as shown in *Figure 3.7*. The work involved in each plan step is divided among *roles*, each of which must be fulfilled by a different agent in an actual situation. Each *role* is associated with its own preconditions, postconditions, and knowledge for each plan step.

Each plan step, s, is represented by the tuple $\langle C, E, A \rangle$. C is a set of conjunctive conditions that describes the *precondition* of s. E is the set of conjunctive conditions that describes the *postconditions* of s. Both preconditions and postconditions may contain negated conditions. A is the *application knowledge* for operationalizating s in different scenarios. The application knowledge, A, is associated with a local casebase that is applicable to only the plan step it is associated with. As *Figure 3.7* depicts, the application knowledge is partitioned among the roles associated with a given plan step and therefore contains only role-specific implementations of the plan step in a variety of situations.

Both the preconditions and postconditions of a plan step are represented by a tuple $\langle V, t \rangle$. V is a list of *perspectives* for each *role* that is responsible for learning that plan step. t is the timeout for how long a precondition or a postcondition should be checked before the agent decides that the condition cannot be satisfied.

Each *perspective* for a role in a plan step is represented by $\langle A, C \rangle$ where A is the name of the role and C is a conjunctive list of conditions. A plan has only one perspective per role, and the perspective of each role describes the conditions that must be satisfied from the local view of the agent that takes on that role.

To demonstrate how perspectives work in plans, let us consider the example in *Figure 3.8* from the soccer domain. Suppose we have a two-step plan that requires three teammates in regions R1, R2, and R3 to pass the ball among themselves starting with the player in region R2. In the first step of the plan, player in region R2 is expected to pass the ball to the player in region R1, and, in the second step, the player in region R1 is expected to pass the ball to the player in region R3.

In order for this plan to be activated, there must be three teammates that happen to be in regions R1, R2, and R3. The player in region R1 will take on role A, the player in region R2 will take on role B, and the player in region R3 will take on role C.

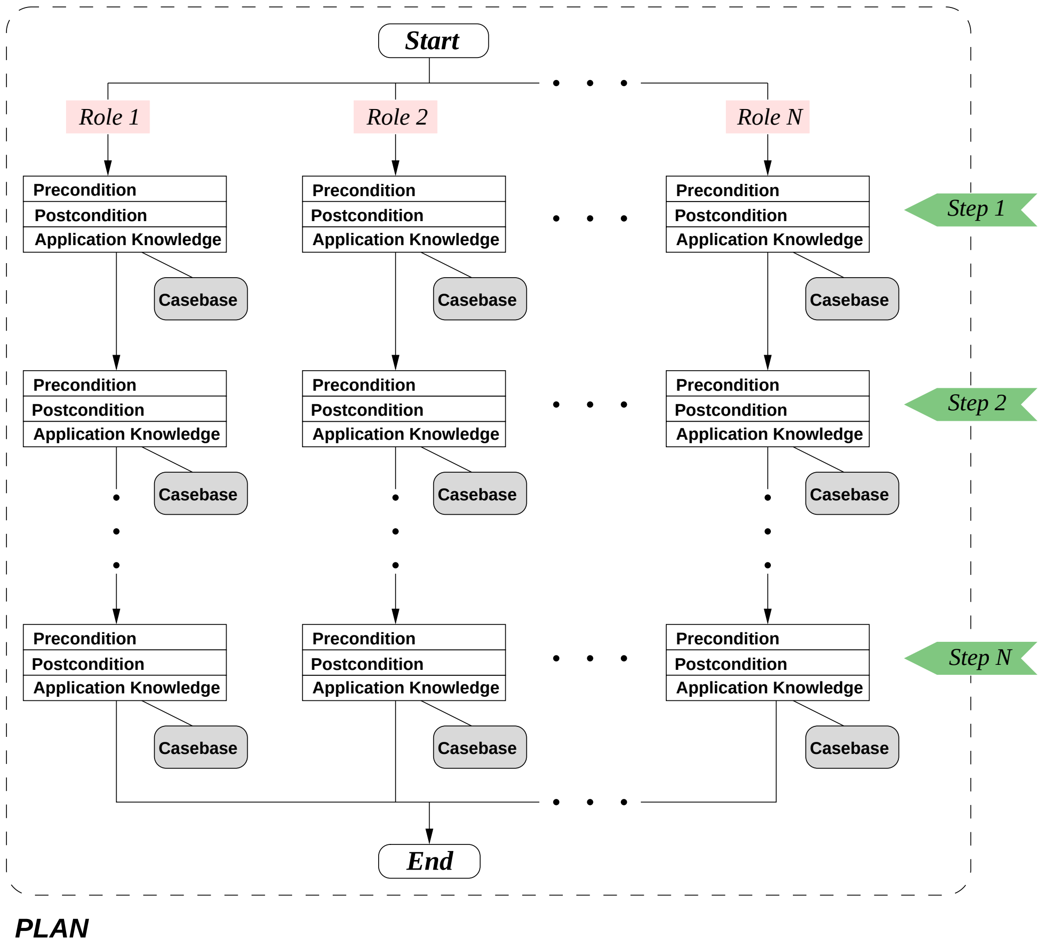


Figure 3.7: Graphical representation of the plan data structure

Therefore, in step 1 of this plan, player B must have the ball and must know that player A is in region R1 and player is in region R3 in order to start executing the plan. Similarly, player A needs to know that there is a teammate with the ball in region R2 who will take on role B and a second teammate in region R3 who will take on role C, while it takes on role A. The player in region R3 must also know that the player in R2 has the ball, and there is a second player that resides in region R1. In the second step of the plan, player C must know that player A has the ball and is in region R1, and player A must receive the ball and know that C is still in region R3 before it can pass the ball to C. As this example demonstrates, each step of a plan is described from the perspective of individual agents such that each agent can know how to contribute to the overall success of a given plan.

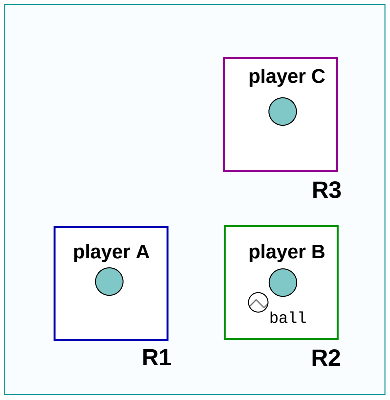


Figure 3.8: An example plan for a three-player plan from the soccer domain where each player needs to check conditions specific for its own role in that plan

Plan step preconditions in MuRAL correspond to STRIPS preconditions. However MuRAL postconditions are treated differently from STRIPS postconditions (add and delete lists). In MuRAL, postconditions specify the completion or termination conditions for a plan step. If the postconditions of a plan step for a role are satisfied, it means that plan step has been successfully applied. If the postconditions time out, it means that the plan step has failed.

Since agents need to operate in dynamic environments that are potentially affected by the actions of other agents (and possibly by the environment), we do not represent plans at the lowest possible detail, i.e., using the primitive actions available in a domain. The reason for this is that each plan step may require multiple reactive actions on the part of each agent involved in executing that plan step, and these actions can, at best, be determined at runtime. Instead plans represent only the critical outcome (i.e., postcondition/goal) expected of a plan step and have each agent decide on the particular reactive actions to implement each plan step at runtime in different circumstances such that the learned cases provide a wider coverage of the external search space involved. To decide on what series of actions each agent should execute to implement a plan step in a particular situation, agents use a form of reinforcement learning. *Section 3.3.5* describes the relationship between search operators and high-level actions that are used to apply plans to different situations, and *Section 3.4.1* describes our learning algorithm.

Case Representation

A case stores role- and situation-specific information about the implementation of a given plan step. The role specification partitions the casebase for a plan step, and the *index* in each case for a given role enables situation-specific matches. This partitioning is important for generality of representation, since an agent may be trained on different roles in a plan. If such training occurs, the agent will acquire cases for each role, and, during application of the learned action knowledge, there needs to be a mechanism to associate cases with each possible role. In our approach, the casebase structure is flat.

The graphical representation in *Figure 3.7* shows only the casebases associated with each role per step. The set of attributes of a case is as follows:

- ***Plan***: The name of the plan the case is associated with.
- ***Step***: The order of the plan step this case operationalizes.
- ***Role***: The name of role that needs to be fulfilled by an agent during execution. The assignment of an agent to a role stays unchanged until that plan is either completed or aborted.
- ***Index***: The description of the *external state*, i.e., the retrieval index of the case.
- ***Success count***: A counter that stores how many times the current case has been successfully used in full-fledged situations.
- ***Failure count***: A counter that stores how many times the current case has been used but failed in full-fledged situations.
- ***Rotation value***: The degrees by which a *normalized* scenario from the perspective of a role had to be rotated during training to get a match. This offset angle value allows the reconstruction of the original match during training such that actions during the application phase can be oriented with respect to the current situation when relative coordinates are used.

- ***Grid unit***: The granularity of the discretization of the physical space into grids. For example, if the grid unit is 3, then all x-coordinate positions within the range $[0\ ..\ 3)$ will belong to x-coordinate grid value 0.

- ***Action sequence***: The action sequence that is acquired during learning. This action sequence solves the subproblem posed by the current plan step for the current role in question. Its usefulness is reinforced by the success rate given by $\text{Success count}/(\text{Success count} + \text{Failure count})$.

3.3.5 Search Operators and Reactive Action Modules (RAMs)

Reasoning with the complex details of a real environment is computationally very inefficient due to the combinatorial growth of the search spaces involved. In order to strike a balance between deliberation and reaction, our approach uses heuristic search to guide (but not dictate in detail) what to do next in a particular situation. Therefore, the operators used during search need corresponding data structures in the *Execution Subsystem* of the agent architecture (*Section 3.3.3*, page 86) so that the deliberative decisions of the agent can be implemented in the target environment.

In our approach, a given search operator in the *Learning Subsystem* corresponds to a program module called a *reactive action module (RAM)* in the *Execution Subsystem*. A RAM is a computer program that implements the high-level goal expressed by a search operator in the actual environment using reactive reasoning so that an agent can respond to dynamic events in realtime. If, for example, a search operator moves the position of an agent by ten units of distance to the left, then the corresponding RAM would implement this navigation task, but the implementation of the RAM would possibly involve functionalities such as obstacle avoidance and contingency handling.

The output of a RAM is a continuous series of primitive actuator commands with ground parameters as depicted in *Figure 3.9*. In the architecture diagram in *Figure 3.6*, this is shown by the wide arrow that joins the *Execution Subsystem* to the *Actuators*, which directly execute these commands. Since a RAM responds to dynamic changes, the sequence of the primitive commands it outputs will vary from situation to situation.

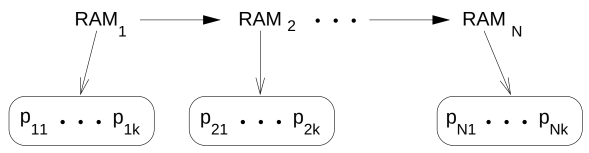


Figure 3.9: The dynamic decomposition of each reactive action module (RAM) into a sequence of primitive actions. Each RAM dynamically generates a sequence of primitive actuator based on current state of the world world.

A RAM may employ other RAMs to handle the contingencies that arise during its execution. Reasoning about contingencies allows each agent to recover from potentially problematic situations and keep its commitments to select to execute. In the soccer domain, for example, a player needs to be close to the ball in order to kick it to another player. If the player is close to the ball but not close enough to kick the ball, a contingency occurs, and the RAM that is used for passing the ball calls the RAM that moves a player to an appropriate kicking position with respect to the ball. When that RAM accomplishes its goal, control returns to the RAM responsible for passing the ball (See *Figure 4.2* in *Section 4.1*, page page 135).

RAMs also attempt to detect catastrophic failures within their contexts, so that an agent does not get stuck in an infinite loop. This way, the control is returned to higher levels of the agent architecture so that the agent may choose a different course of action.

3.3.6 Situation Matching

In applying a plan to a given situation, our system uses two stages of matching. The first stage tries to match the *internal state* of a team of agents to the preconditions and postconditions in each plan step. The second stage tries to find a match between the current *external state* and application knowledge in the casebase of each plan step.

Precondition and Postcondition Matching

The system checks whether preconditions or postconditions of a plan step have been satisfied in the current situation using *rotations*. A *rotation* is a rotated instantiation of the physical scenario defined relative to the *normalized* viewing direction of a role in a given plan step. The normalized viewing direction defines a relative axis for recognizing rotation-dependent patterns in a physical setting.

In the case of soccer, the viewing direction assumed while writing the conditions for a role in a plan step serves as the x-axis of a relative coordinate system whose origin is the center of a given player and whose y-axis is 90-degrees rotated counterclockwise with respect to its origin. The patterns described by a plan condition for a role refer to the constellation of team or opponent players. The *internal state* of a team refers to the constellation of relevant teammates, and, the *external state* of a team refers to the constellation of relevant opponent players.

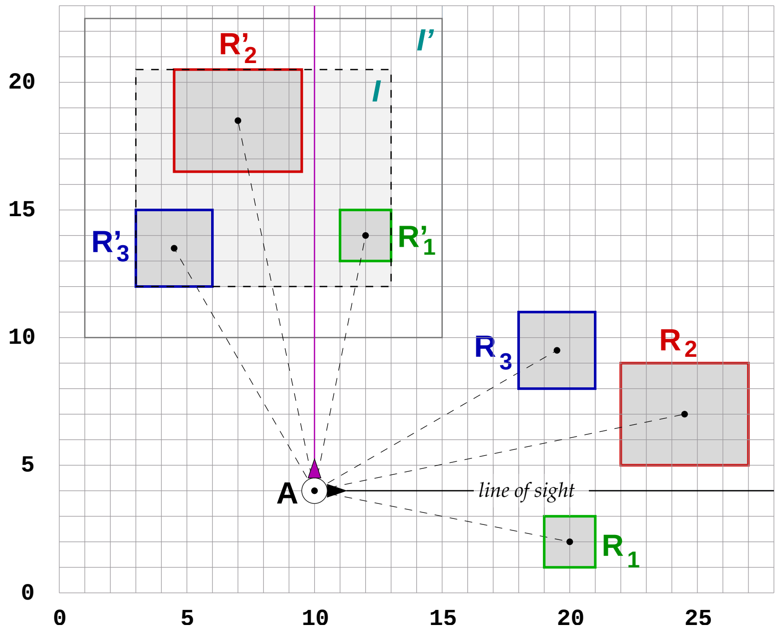


Figure 3.10: An example precondition/postcondition matching using *rotations*. Regions R_1, R_2, and R_3 designate the *normalized* representation of the precondition/postcondition where the relative alignment of agent A's line of sight is the dark horizontal line extending towards the positive-x direction from A's center. A 90-degree counterclockwise rotation of this scenario is represented by regions R'_1, R'_2, and R'_3.

Figure 3.10 shows an example of how rotations are used in matching plans to situations from the perspective of an agent, in this case named A. The rectangles R_1, R_2, and R_3 represent areas in which a plan condition expects to find agents that are teammates of A. In this normalized representation, agent A is looking towards east as indicated by the arrow. This condition can be expressed as

```
((in-rectangle $T_1$ $R_1$) $AND$
 (in-rectangle $T_2$ $R_2$) $AND$
 (in-rectangle $T_2$ $R_3$))
```

where *(in-rectangle t R)* checks whether a teammate agent, t, happens to be within region R. This normalized condition is, however, limited to one particular orientation of the regions with respect to the viewing direction of A. To enable off-view matches, our approach checks all possible orientations of the normalized condition within a given angle range of the current viewing direction at discrete intervals on both sides of the current agent viewing direction. An example of such a possible orientation of the normalized representation is shown in *Figure 3.10*. In the figure, regions R'_1, R'_2, and R'_3 are 90-degree counterclockwise rotated versions of R_1, R_2, and R_3.

For simplicity, regions themselves are not rotated. Instead, only their middle point is rotated. For example, region R_2 is 5 units long and 4 units wide. Its rotated version, R'_2 is also 5 units long and 4 units wide. Note also that the condition specifications are in terms of actual coordinates and not in grid coordinates.

Case Matching

Case retrieval in our system is preceded by the matching of the preconditions of the corresponding plan step to the internal state of a multiagent team. Then, in a given situation, a case is selected for application from the local casebase associated with that plan step based on the external state of the multiagent team. Therefore, the matching of case indices to situations takes the matching of preconditions as a starting point.

To determine which non-team members to consider as part of the external state description, we construct an *influence area* of the matching rotation. An *influence area* is the smallest rectangular region that contains all teammate regions specified in a plan step condition. For example, if we suppose that agent A in *Figure 3.10* found a match for the 90-degree rotation $\{R_1, R_2, R_3\}$, then the influence region of this rotation would be the area labeled I. A strip of area around this *influence region* is very relevant for successfully executing the current plan step, since the existence of other agents may potentially affect the desired outcome. Therefore, to include the relevant region around the *influence area*, we consider an expanded version of I, which is marked as I' in the figure. Then, the agents in region I' that are not teammates of agent A will constitute the external state description for this situation. Once the external state description is obtained, then A can compute the similarity between cases in its local casebase for the current plan step and the external situation description.

After similarity computation is completed, A has to select the best match. However, multiple cases may be equally similar to the current situation, and favoring the case with the highest success to failure ratio would not give a chance to other similar cases. It is important to note that the application of cases involves reinforcement learning, and, to have wide coverage of the domain, all similar cases in a situation must have a fair chance of being selected.

Therefore, we employ a probabilistic selection scheme to select a case from among multiple similar cases. The details of this selection scheme are discussed in *Section 3.4.3*.

For matching cases to situations, we use the external conditions of the environment. In the case of soccer, external conditions refer to the constellation of opponent players. Matching at this level happens using *influence regions* around each opponent player. An influence region is a discrete rectangular region centered at the player's current location and specifies weights that specify a "strength" of the influence of the player's presence at a given grid distance. At grid distance d, the strength weight of a grid, $f(d)$, is given by *Equation 3.1*. When the grid distance is larger than 2, the strength is taken to be zero.

$$f(d) = \begin{cases} 0.5 - 0.75d + 2.25d^2 & \text{if } (d \leq 2) \\ 0.0 & \text{otherwise} \end{cases} \tag{3.1}$$

The *influence* of a given player on a grid at distance d, $y(f(d))$, is given by *Equation 3.2* such that the farther a grid square, the smaller the strength of the influence. For example, for $d = 2$, the influence region of an opponent player will have the weights shown in *Figure 3.11*.

$$y(f(d)) = \frac{1}{f(d)} \tag{3.2}$$

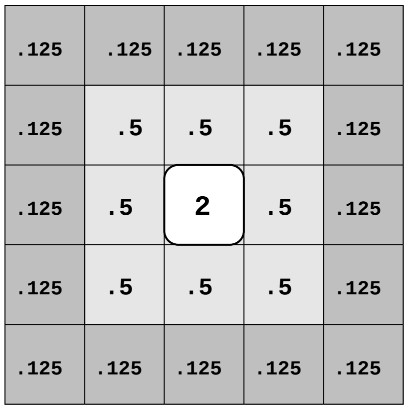


Figure 3.11: Influence region of a player in the areas around its position. The grid square in the middle represents where the agent is currently located. The influence of the player decreases. Beyond a certain grid distance, the influence becomes zero.

Since matching cases to situations involves comparing the opponent player constellation in each case to that in the current setting, we sum the influences of both such that overlapping regions get added. Then we take the difference of the combined influences at each grid location. Finally, we sum the differences to get a measure of the difference between the case and the current situation.

To "normalize" comparisons across different types of cases, we compute the average of the difference by dividing the total difference by the number of the opponent players in the current situation. To account for the difference in the number of players, we introduce a penalty value that is multiplied by the difference in the number of opponent players in a case and the situation being considered. This penalty is the difference we get if the influence region of a player in a case is not matched by any players in the current situation. That is, the penalty is the addition of all the weights in the influence region of a single player, as in *Figure 3.11*. Then the difference (or inverse similarity) of a case is the sum of the average difference and the player difference penalty.

To decide if a given case is similar to the current situation, we compute the worst difference we can tolerate. This is given by the penalty value mentioned above multiplied by the number of opponent players in each case. If the difference of a case is less than or equal to the maximum difference, that case is considered similar.

3.3.7 Optimality vs Efficiency in Heuristic Search

Optimality is a main issue in theoretical studies of multiagent systems. In complex systems, on the other hand, optimality becomes very difficult to define and achieve. Therefore, multiagent systems that operate in realistic or real environments aim to solve problems efficiently but not necessarily optimally (e.g., [Balch, 1997b; Matarić, 1998; Stone and Veloso, 1999; Briggs and Cook, 1999; Bonet and Geffner, 2001b])

The main reason why optimality is not a main issue in complex domains is the unpredictability of such domains. In order for agents to design plans that involve relatively long action sequences that span potentially many state transitions, an agent needs to be able to predict how the environment and other agents will behave. To predict the actions of other agents, an agent has to have a model of the decision mechanisms of other agents. However, even then, the interdependencies among agents can cause infinite regression.

If the action sequencing problem in a multiagent system is to be solved using heuristic search, an agent can search using all possible combinations of agent actions. There are two problems with this approach. The first problem is that combining actions of all agents does not produce a meaningful representation of what may happen in the actual environment. In a real world situation, actions of different agents may be interleaved in many ways since actions are durative and agents behave asynchronously. Therefore, considering all possible interleavings is impractical, particularly because of the increased search space. The second problem is that an agent cannot assume an order on all other relevant agent actions when searching. That is, an agent cannot reliably predict what other agents will do so that it can choose the best action possible to execute next. If we make the assumption that all agents are rational, it is also impractical to assume that each agent is capable of rationalizing the behavior of all other relevant agents from their own perspectives of the world. We cannot assume that an agent can observe the world such that it can know what other agents currently know. In addition, an agent cannot always know the decision mechanisms or the preferences of other agents to duplicate their reasoning, and neither can it predict when other agents will act.

If an agent uses its own decision mechanisms and preferences as an approximate model of other agents, it will be conducting a type of multi-player minimax search to make decisions about what to do next. However, unlike in games, an agent operating in a complex environment cannot afford to wait for other agents to make their moves so that it can maximize its next move since there is no synchronization or ordering of moves. Instead, to make progress towards satisfying its goals, an agent has to plan short sequences of primitive actions based on the current or, at most, the recent set of states of the world. Hence, in the MuRAL approach, we assume that the world is temporally static for a brief while into the future so that an action sequence can be found that may work in the actual environment despite dynamic future changes. Also the agent cannot make assumptions about how other agents will behave in a utility maximizing way during the brief period for which it is planning using search. In summary, optimization of actions in complex environments is difficult and impractical to define. As in other algorithms applied to complex domains, the decision mechanisms we implemented strive for efficiency rather than optimality.

3.3.8 Hierarchical Representation of State Spaces

In MuRAL, we divide the state space hierarchically by distinguishing the conditions that a collaborative group of agents can potentially affect from those that it does not or cannot have control over. We refer to the first type as *internal conditions* or *internal state*, and we refer to the second type as *external conditions* or *external state*. The rationale behind this distinction is to enable a multiagent system to select plans at runtime based on more controllable internal conditions without having to reason about the remaining less controllable dynamic details of a situation, that is, the external conditions. Then, for execution, each agent in the multiagent system chooses a solution to apply based on the external conditions. So we treat the internal state almost as a constant and the external state as varying around a ground internal state description. Hence, this is a hierarchical method of filtering in available action options. In the physical world, the relative layout of the agents in a group can be an important motivator for choosing which plan to apply, and the remaining conditions become details to be worked around. In this document, we refer to the physical layout of a group of agents as a *constellation*.

So, for a given set of internal conditions, there can be many sets of external conditions that need to be handled. The constellation of a team of agents then become part of the internal state of that team, and the constellation of non-team agents in the same environment become part of the external state. Since we take internal conditions as basis for choosing plans, a plan is written only based on internal conditions. Then the learning component of the system deals with how each agent in a multiagent plan should act in order to successfully execute each plan step under different external conditions. This hierarchical relationship between internal and external conditions is shown in *Figure 3.12*, where each parent internal condition set is common to all external condition sets associated with it such that each external condition set forms a unique *precondition group* with its parent internal condition set. For example, $\{I_1, E_{12}\}$, $\{I_2, E_{2N}\}$, and $\{I_M, E_{M2}\}$ are precondition groups that uniquely identify situations about which an agent would have learned possibly many solutions.

In the game of soccer, for example, internal conditions would include the constellation of a group of players that are teammates, and the external conditions would include the constellation of opponent players. If the attackers happen to be positioned such that a certain soccer tactic can be used, it would be desirable to have different variations of applying that tactic such that different opponent constellations can be handled but the same overall tactic is operationalized.

Our hypothesis is that, if each agent in a multiagent system learns how to handle specializations of the external state from its perspective, keeping the current internal conditions fixed, it can apply each step of a given plan successfully, thereby successfully applying the entire plan. In addition, if the agent learns which of the previously learned operationalizations of a plan step are most effective, it can learn better implementations of each plan step to execute.

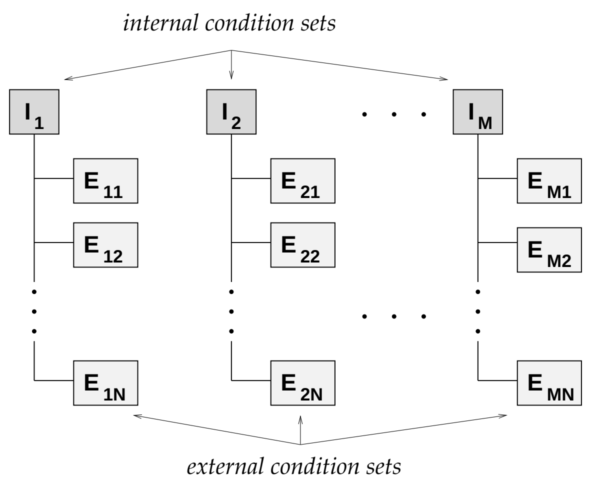


Figure 3.12: Hierarchical relationship between *internal* and *external* conditions that describe the world state

3.3.9 Discretization of State and Action Spaces

Continuous state and action spaces in complex domains are of infinite size in their natural form. Even after discretization, search spaces can still be too large to search exhaustively or heuristically.

For example, the simulated soccer domain is complex and has a very large state and action spaces. If we divide the field into 1 unit-distance grids, the 22 players and the ball can have a total of $(105 \times 68)^{23}$ possible locations. With player viewing direction added and each dimension discretized in one-tenth steps, each agent could be in any of the $(1050 \times 680 \times 3600)$ unique orientations [Stone, 1998, Chapter 2]. If we consider the actions that are available to all non-goalie players `turn`, `turn_neck`, `dash`, and `kick`, with somewhat coarse discretization, an agent still has about 300 different instantiated actions it can take every cycle [Riedmiller et al., 2002]. So, if we discretize angles/directions at 10-degree intervals and power-related arguments at 5 power-unit intervals (maximum power value being 100), we will have the following number of instantiations for each type of action:

- `turn( angle )`: $360/10 = 36$ action instantiations
- `turn_neck( angle )`: $180/10 = 18$ action instantiations
- `dash( power )`: $100/5 = 20$ action instantiations

- `kick( power, direction)`: $(100/10 = 10) \times (200/10 = 20) = 200$ action instantiations

Therefore, for each team, there are 274^{11} different action instantiations at each decision cycle. With finer discretization, the number of action options could be as large as 300^{11} given in [Riedmiller et al., 2002] or much larger. Therefore, limiting the size of both the search and action spaces is the first requirement in problem reformulation in order to be able to generate solutions to complex problems.

In MuRAL, we also discretize the physical space where agents operate into grids. A grid is a square region of a certain dimension that allows the search space to be reduced. This grid representation plays an important part in the reasoning and learning procedures of an agent, but agents execute their actions in a continuously-valued environment. That is, we do not assume that an agent can only move between grids.

Figure 3.13 depicts an example case where a grid laid over a physical region represents the discretized positions of two sets of agents ({A, B, C} and {D, E, F, G}).

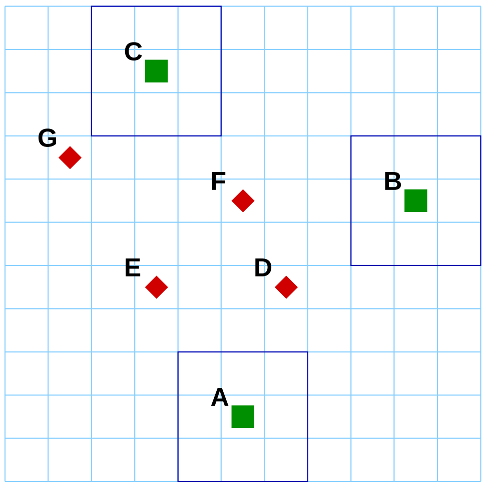


Figure 3.13: An example grid laid over a region with two teams of agents

The search process for discovering action sequences to apply to a situation and the cases learned by the system are represented in terms of this grid representation. The action space is also discretized. With its specific parameters aside, an action in the physical world can be represented in terms of an angle. In a grid world, there are eight possible direction for an action as *Figure 3.14* depicts.

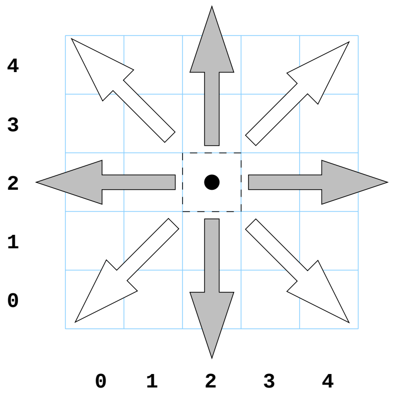


Figure 3.14: Eight possible directions for an action in a discretized world

Because of our approach that separates execution from planning, we do not need to use all eight directions since any grid is reachable from any other grid by moving in the four non-diagonal directions (north, south, east, west). Therefore, for action sequence search, the branching factor becomes 4 instead of 8.

3.3.10 Plan and Plan Step Instantiation

The instantiation of a plan requires the assignment of roles in that plan. This instantiation takes place when an agent matches a plan to the current situation. Action sequences stored in cases as solutions to the scenario described by that case are normalized (See *Section 3.3.6*). Plan step instantiation requires that this action sequence be instantiated for the current situation. This involves rotating any direction-dependent actions in the normalized solution definition in the retrieved case for the current situation.

3.4 Algorithms

This section describes the main algorithms used in this dissertation.

3.4.1 Learning Algorithm

Algorithm 3.1 gives the learning algorithm, `Learn`, each agent uses during the training and application phases of our system. It is used both for learning a new case about how to operationalize a given plan step as well as for applying existing cases to current situations, thereby acquiring positive or negative reinforcement on the plan and its steps when naïve reinforcement learning is activated.

The learning algorithm, `Learn`, requires eleven inputs:

- A: The agent that is responsible for learning about a given plan step
- P: The plan about which agent A is supposed to acquire knowledge
- $step$: The ordinal index of the plan step
- R: The role that agent A will fulfill while learning about plan step $step$ in P
- $doSearch$, $doRetrieval$, $doRL$, $doNewCaseRetain$: These four parameters control the operation mode of the `Learn` algorithm. They turn on/off whether the agent should do search, retrieval, reinforcement learning, and case retention in a given version of the system. The *doSearch* parameter controls whether the learning algorithm calls its heuristic search facility to discover a sequence of actions to handle the current problem posed by plan step *step*. The *doRetrieval* parameter controls an agent's access to the knowledge in that agent's plan casebases. The *doRL* parameter controls whether an agent will modify the failure and success counts in the plan steps it applies. The *doNewCaseRetain* parameter controls whether an agent will save cases based on its experiences. During training, for example, *doSearch* and *doNewCaseRetain* are turned on. During application mode with naïve reinforcement learning, on the other hand, *doRetrieval* and *doRL* are turned on.
- E: The description of the external conditions that further define the context for learning. The external state description is derived as discussed in *Section 3.3.6*, page 96.
- u: The unit for discretizing the physical space in which agent A is expected to operate. Every u units of distance in the actual environment correspond to 1 unit of grid distance in the discretized version of the environment.

```
Learn( Agent A, Plan P, int step, Role R, bool doSearch, bool doRetrieval,
       bool doRL, bool doNewCaseRetain, ExternalState E, GridUnit u,
       int maxSearchDepth )
   C=null;
   if (doRetrieval)
      C = RetrieveCase( P, step, R, E ); // See Algorithm 3.3, pp. 113
   end if
   if (C)
      S = C.getSolution();
      success = A.applySolution( P, step, S );
      if (success)
         if (doRL)
            C.successCount++;
         end if
      else
         if (doRL)
            C.failureCount++;
         end if
      end if
   elsif (!doSearch)
      S = {} // Empty solution
      success = A.applySolution( P, step, S );
   else  doSearch=true
      K = P.buildSearchProblem( A, step, P, u, E );
      N = bestFirstSearch( K, maxSearchDepth );
      if (N)
         S = N.extractSolution();
         success = A.applySolution( P, step, S );
         if (!success) return $FAILURE$;
         elsif (S is empty) return $SUCCESS$; end if
         if (doNewCaseRetain)
            t = P.conditionSetMode( R, step );
            if (t is not absolute coordinate mode)
               rotate E by -P.matchingRotationAngle
            end if
            newE = {} // Empty
            if (S is dependent on E)
               if (t is not absolute coordinate mode)
                  Copy relative position of each entry in E to newE;
               else
                  Copy global position of each entry in E to newE;
               end if
            end if
            C = buildNewCase( P, R, step, newE, S, u );
            P.addNewCase( R, C );
         end if
      else
         return $FAILURE$;
      end if
      if (doRL or doNewCaseRetain) A.storePlan( P ); end if
   end if
   return $SUCCESS$;
end
```

Algorithm 3.1: Learning algorithm

- $maxSearchDepth$: The maximum depth to which the heuristic search the learning algorithm uses will explore possibilities

The learning algorithm, `Learn`, works as follows:

- On line 2, the algorithm initializes case C to null to prepare for potential retrieval.

- If parameter $doRetrieval$ is $true$ (line 3), then the agent will try to apply already learned knowledge about plan step $step$ in P. Given the plan step, $step$, current role of the agent, R, and the current external state description, E, the `Learn` algorithm tries to retrieve a case that matches E (line 4). This case is stored in C. If no cases are retrieved, the algorithm moves to the if-statement starting at line 18, where the *applySolution* function is called. Given a solution either retrieved from the case or one discovered using search, *applySolution* applies that solution to the current situation and checks whether the postconditions of the current plan step ($step$) have been satisfied until the postconditions time out.

  If the algorithm moves to line 18 after the if-statement at line 6 fails, then, by definition, there is no solution the algorithm can apply. Hence, S is empty (line 19). However, the algorithm still needs to monitor the postconditions of the current plan step. Since the *applySolution* accomplishes this task, we call it with an empty action sequence (line 20).

- If, on the other hand, there is a matching case (line 6), the algorithm accesses the action sequence, S, stored in the retrieved case C (line 7).

- Then the algorithm starts executing this action sequence in the current situation (line 8).

- If the application of the action sequence S is successful (line 9) and naïve reinforcement learning mode is active (line 10), then plan step $step$ receives positive reinforcement (line 11).

- If the application of S fails, C receives negative reinforcement (line 15).

- At line 21, the algorithm starts the heuristic search mode. At this point, *doSearch* must be true.

- In search mode, the algorithm builds a new search problem based on the state of the current agent, A, the plan step in question, the current external state description, and the discretization unit, u. The algorithm calls *buildSearchProblem* on the current plan P (line 22). The return value of this call, K, represents a data structure that defines a search problem to solve.

- Then the algorithm tries to solve the search problem using best-first heuristic search by calling *bestFirstSearch* (line 23). The return value of this call, N, is a search node, from which the action sequence that solves the search problem is instantiated using the *extractSolution* call (line 25) given that search produced a result (line 24).

- If search fails to find a solution till depth *maxSearchDepth*, the algorithm fails and returns with failure (line 46).

- If search returns an action sequence, that action sequence S needs to be tested in the actual environment to find out whether it will work in practice or not (line 26). For this, the algorithm calls *applySolution* and passes S to it (line 26). The *applySolution* call executes S. The return value of this call, *success* indicates if S worked in the actual environment or not.

  How the *applySolution* call translates each action in S to an actuator command in the agent architecture is critical to how a MuRAL agent operates (see *Figure 3.6*, page 87).

- If S works in practice (line 26), then it means the agent can store a new case as long as *doNewCaseRetain* is turned on (line 29). If not, the algorithm returns with a failure (line 27). If S happens to be empty, then there is no other task to perform within `Learn`, and the algorithm returns with success (line 28).

- In building a case, it is important to distinguish whether the postconditions of the current plan step, *step*, were written in terms of global coordinates or relative coordinates. Therefore, first, the algorithm determines the type of the current postconditions (line 30). If the postconditions for role R in the current plan step are in terms of relative coordinates (t), the algorithm rotates the current solution by inverse of the matching rotation angle to "normalize" the solution (line 32).

- Lines 34–41 deal with the storage of the external conditions, E. The external conditions are stored as part of a case if and only if the solution S is dependent on them. Since E is the determinant in matching cases to situations, we wish to retain the description of the external environment when that information is needed to match solutions to situations. If S is dependent on E (line 35) and the postconditions are in terms of relative coordinates (line 26), the algorithm copies the relative coordinates of the objects in E to an initially empty $newE$. If t indicates global coordinates, then the algorithm copies the global coordinates of the objects in E to $newE$.
- On line 42, the algorithm builds a new case to store in the casebase of agent A, given information about the current plan, P, the role of the current agent, R, the external state description at the time the search problem was built, $newE$, the action sequence, S, and the unit of discretization, u. The newly built case, C, is then stored as part of P at step $step$ for role R (line 43).
- On line 48, the algorithm stores the modified version of the plan to an output file. This is done only during learning and training.
- Finally, on line 50, the algorithm returns with success, since this point can only be reached if there were no failures up to that point.

3.4.2 Training Algorithm

The algorithm each agent uses during the training phase is given in *Algorithm 3.2*. Parts of this algorithm are specific to the RoboCup soccer simulator but could be generalized to other simulator domains.

Training algorithm, `Train`, requires four inputs:

1. A: The agent that is being trained

2. P: The plan that the agent will train on

3. u: The discretization unit for the physical space in which the agent operates. Every u units of distance in the actual environment correspond to 1 unit of distance in the discretized version of the actual environment

4. $maxSearchDepth$: The depth limit to the search for finding a sequence of actions for satisfying the goal of a plan step

```
01: Train( Agent A, Plan P, GridUnit u, int maxTries, int maxSearchDepth )
02:    foreach plan step index i in P
03:        s = P.getPlanStep( i );
04:        match=false;
05:        while (s.preconditions have not timed out)
06:           if (A.actionAgenda is empty)
07:               Add default action to A.actionAgenda;
08:           end if
09:           match = P.checkPreconditionMatch( A, s  );
10:           if (match)
11:               break; // Done checking preconditions
12:           end if
13:        end while
14:        if (match)
15:           E = A.getCurrentExternalState();
16:           R = P.getMatchingRole( A );
17:           success = Learn( A, P, i, R, true /*doSearch*/,
                               false /*doRetrieval*/, false /*doRL*/,
                               true /*doNewCaseRetain*/, E, u,
                               maxSearchDepth );
18:           if (not success)
19:               return FAILURE;
20:           end if
21:        else
22:           return FAILURE;
23:        end if
24:    end for
25:    return SUCCESS;
26: end
```

Algorithm 3.2: Training algorithm

The training algorithm, `Train`, works as follows:

- The main body of the algorithm (lines 2–24) is composed of one loop that iterates over each plan step in the given plan and trains the agent. The first operation in training is to check if the preconditions of the current plan are satisfied (lines 5–13). The algorithm checks, in an inner **while** loop, whether the preconditions of plan step i have been satisfied and whether they have timed out. Both preconditions and postconditions are associated with a timeout value. [2] If a certain number of attempts fails, then the condition is said to have not been satisfied, and the algorithm returns with failure.

- On line 3, the algorithm retrieves the ith plan step and stores it as s. A plan step is a compound data structure that contains information about roles and other information such as timeout values for satisfying conditions. (See *Section 3.3.4*).
- If the Action Agenda of the agent is empty, the algorithm adds a default action so that the agent can monitor its environment and respond to basic situations such as an incoming ball (lines 6–8).
- On line 9, the algorithm calls the $checkPreconditionMatch$ function to check for precondition match. If there is a match (line 10), then the algorithm exits the inner **while** loop. If there is no match and the preconditions time out, the algorithm returns with failure (line 22). The checking of the preconditions for the very first plan step is a special case that needs to be handled differently from subsequent steps. The checking of whether the preconditions of the first step in a plan are satisfied or not requires that preconditions for all roles associated with that step be tested, since there is no role assigned to the agent in training.
- On line 15, the `Train` algorithm collects a description of the external conditions by calling $getCurrentExternalState$ and stores those descriptions in E.
- On line 16, the algorithm accesses the role that agent A has assumed in plan P. Checking for precondition match during the very first plan step assigns the roles of each agent involved in executing the plan. Each agent assigns roles independently and does not communicate its assignments to its collaborators. So, a plan will be

[2]In our implementation, we determined the values of these timeouts emprically.

executed successfully if the role assignments of all collaborating agents happen to be identical. In addition, an agent's role, R, remains the same throughout P.

An agent takes on the role associated with the first perspective whose preconditions it satisfies. It is, however, possible that more than one training agent can match the same role. However, this is resolved practically during training and application by how collaborating agents are placed on the field, so that their constellation can easily match the preconditions and the postconditions in a plan.

- On Line 17, the `Train` algorithm calls the `Learn` algorithm in *Algorithm 3.1*. If this call does not succeed, the algorithm returns with failure (line 19). To enable the learning algorithm to work with the training algorithm, search and case retention modes are turned on in the `Learn` algorithm call. Note that the `Learn` algorithm internally checks whether the postconditions of the current step have been satisfied.

- If the postconditions have been satisfied, $success$ is set set to $true$ (line 17), and, therefore, the agent continues training on the next plan step in the plan. If there are no plan steps left, then $sucess$ is set to $false$ (line 17), and this causes the algorithm to exit with failure (line 19). As in the case of preconditions, postconditions for plan step i are checked only from the perspective of agent A that took on role R in plan P.

3.4.3 Case Retrieval Algorithm

The case retrieval algorithm is straightforward, and it is given in *Algorithm 3.3*.

The case retrieval algorithm, `RetrieveCase`, requires four inputs:

1. P: The plan that the agent matched to the current situation and assumed a role in

2. $planStep$: The numerical index of the plan step about which a case is to be retrieved from the current plan

3. R: The role the agent took on to execute plan P

4. E: The description of the external state. This description serves as the index into the casebase associated with role R

```
01: RetrieveCase( Plan P, int planStep, Role R, ExternalState E )
02:    s = P.getPlanStep( planStep );
03:    B = s.getCasebaseForRole( R );
04:    p = s.getPostconditionsForRole( R );
05:    if (not B)
06:       return null;
07:    end if
08:    foreach case c in B
09:       c.computeCaseSimilarity( E );
10:    end foreach
11:    RandomCaseSelector S={}
12:    foreach case c in B
13:       if (c is similar to E)
14:          add c to S;
15:       end if
16:    end foreach
17:    return S.selectCase();
18: end
```

Algorithm 3.3: Case retrieval algorithm

The case retrieval algorithm, `RetrieveCase`, works as follows:

- The algorithm retrieves the plan step indicated by index $planStep$ and stores this data structure in s (line 2). On line 3, the algorithm accesses the casebase for role R in step s of plan P and stores a reference to the casebase in B. The algorithm also accesses the postconditions of step s for role R. A reference to the postconditions is stored in p.
- If there is no casebase for R in s, then the algorithm returns with an error (lines 5–7).
- Then, for each case c in B, the algorithm computes the similarity of c to the current situation. The similarity of each c is computed based on the retrieval index, E (lines 8–10).
- Using the results of the similarity computation (lines 8–10), the algorithm then determines which cases in B are similar to the current situation (lines 12–16). The algorithm then enters each similar case with its similarity value to a random event selector, S, which is initialized to an empty list.
- Finally, on line 17, the algorithm randomly selects a case from S, where the probability of selecting each case c is a function of the similarity of c to the current

situation computed at line 9.

The random selection of cases works as follows. First, the goal is to select each case with a probability that is strongly related to its success rate. At the same time, we do not want to completely ignore cases with low success rates, especially at the beginning of naïve RL learning. Therefore, we add a contribution factor to the success rate of each case that is proportional to the number of similar cases entered into the random case selector divided by the total number of trials in all similar cases. The effect of this contribution factor is that, as the number of total trials gets large, the overall weight of cases with zero success rate will decrease.

Given a set of n cases that have been deemed similar (line 13 in *Algorithm 3.3*), S, the raw success rate of each case, c_i in S is:

$$rawSuccessRate_{c_i} = \frac{successCount_{c_i}}{successCount_{c_i} + failureCount_{c_i}}$$

The total number of times the cases in S have been applied is given by:

$$totalTrials = \sum_{i=1}^{n}(successCount_{c_i} + successCount_{c_i})$$

The total raw success rate for S is:

$$totalRawSuccessRate = \sum_{i=1}^{n} rawSuccessRate_{c_i}$$

The *contribution* of each case c_i is given by:

$$contribution = \frac{n}{totalTrials}$$

The total rate is given by the sum of the total raw success rates and the total contribution of all cases in S:

$$totalRate = totalRawSuccessRate + (contribution * n)$$

Finally, the probability of selecting each c_i is given by:

$$Prob(selecting\ c_i) = \frac{rawSuccessRate_{c_i} + contribution}{totalRate}$$

Consider the following example, where we have five cases with (success, failure) values of S={(0, 0), (0, 0), (1, 2), (2, 3), (0, 0)}. Since the total number of trials is only 8 (1+2+2+3), the cases that have not yet been used for application still have a relatively high chance of being picked, while cases that have already been used are assigned higher probabilities of selection. [3]

```
c1: raw success rate=0 (0/0)          prob( selection )=0.145631
c2: raw success rate=0 (0/0)          prob( selection )=0.145631
c3: raw success rate=0.5 (1/2)        prob( selection )=0.262136
c4: raw success rate=0.666667 (2/3)   prob( selection )=0.300971
c5: raw success rate=0 (0/0)          prob( selection )=0.145631
```

The second example demonstrates what happens as the number of total trials in a set of similar cases increases. For the input set, S={(1, 5), (12, 34), (0, 0), (1, 2), (1, 1), (0, 0), (0, 0)}, we get the following probabilities of selection. As opposed to the first example above, cases without trials get picked with substantially less probability.

```
c1: raw success rate=0.2 (1/5)        prob( selection )=0.110832
c2: raw success rate=0.352941 (12/34) prob( selection )=0.163342
c3: raw success rate=0 (0/0)          prob( selection )=0.0421642
c4: raw success rate=0.5 (1/2)        prob( selection )=0.213833
c5: raw success rate=1 (1/1)          prob( selection )=0.385501
c6: raw success rate=0 (0/0)          prob( selection )=0.0421642
c7: raw success rate=0 (0/0)          prob( selection )=0.0421642
```

3.4.4 Plan Application Algorithm

After a plan is matched by an agent, an agent starts applying that plan to the current situation. We must note that, by this time, the agent has already checked that the preconditions of the first plan step in the plan have already been satisfied.

[3] If the input raw success rate is undefined, we adjust the value to zero.

The plan application algorithm, `Apply`, is given in *Algorithm 3.4*, and it requires seven inputs:

- A: The agent that will apply the given plan
- P: The plan that the agent will apply in the current situation
- u: The discretization unit for the physical space in which the agent operates. Every u units of distance in the actual environment correspond to 1 unit of distance in the discretized version of the actual environment
- $doSearch$, $doRetrieval$, $doRL$, $doNewCaseRetain$: These four parameters control the operation mode of the `Learn` algorithm (See *Algorithm 3.1*).

```
Apply( Agent A, Plan P, GridUnit u, bool doSearch, bool doRetrieval,
       bool doRL, int maxSearchDepth )
   R = P.getMatchingRole( A );
   foreach plan step index i in P
      s = P.getPlanStep( i );
      while (s.preconditions have not timed out)
         match = P.checkPreconditionMatch( A, s  );
         if (match) break; end if
         if (s.preconditions timed out)
            return $FAILURE$;
         end if
      end while
      E = A.getCurrentExternalState();
      success = Learn( A, P, i, R, doSearch, doRetrieval,
                       doRL, $false$ /*doNewCaseRetain*/,
                       E, u, maxSearchDepth );
      if (not success)
         return $FAILURE$;
      end if
  end foreach
  return $SUCCESS$;
end
```

Algorithm 3.4: Plan application algorithm

The plan application algorithm works as follows:

- First, the role that agent A has taken on in executing plan P is accessed from the plan data structure and is stored in R. A matching plan is one whose preconditions of its first step have already been satisfied by the current *internal state* of the collaborating team of agents who do not communicate.

- The rest of the algorithm is a loop that iterates over each single step, s, of plan P and tries to execute that step successfully in the current environment (lines 3–19).
- With this loop (lines 5–11), a second inner checks whether the preconditions of plan step s have been satisfied by calling $checkForPreconditionMatch$ (line 6). If there is a match, the inner **while**loop is exited. If, on the other hand, the preconditions time out, the algorithm returns with failure (lines 8–10).
- On line 12, the algorithm dynamically collects a description of the current external state and stores it in E.
- Next, the `Apply` algorithm calls the `Learn` algorithm (*Algorithm 3.1*) to do the rest of the work of applying the plan to the current situation.
- A plan step is successfully executed, when the `Learn` algorithm can find a case that contains an action sequence that successfully executes in the current context (line 13). If this happens, then the next step is to check whether the executed action sequence has indeed satisfied the postconditions of the current plan step. This is done internally by the `Learn` algorithm. If the call to `Learn` returns true ($success$, then it means that the current plan step has been successfully executed as intended by the plan specification. If the return value $success$ false, it means that even a successful execution of the action sequence has not satisfied the postconditions of the current plan step. Therefore, the algorithm returns with a failure (lines 16–18).
- If all plan steps are successfully applied to the current situation, then the algorithm returns with success (line 20).

3.5 Evaluation Method

In complex, dynamic, and uncertain environments, optimal solutions are generally not known. For that reason, evaluation of systems that operate in such environments using domain-independent metrics is very difficult [Matarić, 1996]. Instead, researchers use largely domain-dependent evaluation metrics. We face the same evaluation challenge in this research. When optimal solutions are unknown, it is at least desirable to use evaluation metrics that can be standardized across different test iterations such that they indicate the improvement in performance over time. In MuRAL, we used the Soccer Server simulated soccer environment [Kitano et al., 1997b,a; Chen et al., 2002] for all development and testing, and our goal is to show that the system *learns* and improves its performance with experience compared to its non-learning versions.

Since an agent in MuRAL learns in both the training and application phases, we have two types of learning. The learning in the training phase aims to help with *coverage*, and the learning in the application phase aims to improve the selection of plans and plan step implementations by increasingly (but not totally) favoring the previously successful ones to increase the chance of success in future situations. An agent can be trained any time in MuRAL so that it can acquire new cases to help increase the coverage of its application knowledge and also learn to implement new plans. The number of cases acquired to implement a plan, however, is not a good indicator of performance by itself, since, without the reinforcement knowledge, an agent cannot know if a case it selects will work in the application phase. Therefore, both types of learning are required for defining true coverage.

To evaluate the performance of learning, we compare the *learning* version of our system to two other versions of the same system over a relatively large number of experiment trials. The *retrieval* version of our system uses the cases retained during training but does so before any reinforcement takes place. The *search* version of our system does not use any cases from training but instead dynamically searches for solutions at every step of a given plan as an agent would during training. Since the search version of our system operates at the lowest level among the three, we use it as our evaluation baseline. We do not use a random or manual programming approach as baseline. In theory, it is possible to generate action sequences randomly or program the agent behavior manually. In the context of our research, both of these options are not practical. MuRAL agents find solutions during training using high-level search operators (*Section 3.3.2*). Some of these operators take on continuosly-valued arguments. Since MuRAL agents generate these solutions *in situ*, the argument values reflect a potential solution for future situations. Attempting to generate sequences of actions with working parameters and doing so over several plan steps in collaboration with other agents would be impractical. Similarly, manually programming the implementation of each plan for many different situations would be an overly arduous task due to the variability in agent behavior. Automating the adaptation of behavior via learning, we contend, is a more effective than any random or manual approach. The goal of our evaluation approach is to show that:

1. learning (via training) is better than behaving reactively
2. learning via naïve reinforcement improves the performance of the knowledge retained during training
3. "to know" is better than "to search"

During training, each agent has to do *search* to determine its solution at each plan step. However, an agent cannot, in general, know whether the solutions it discovers through search will have eventually contributed to the successful application of a plan. In *retrieval* mode, an agent does not do search but instead retrieves cases that are known to have contributed to the overall success of a given plan during training. In *learning* mode, the agent builds experience about which of the cases it can use to apply a given plan step have been successful so that it may be able to make better choices about which case to pick for each situation. Therefore, we are essentially comparing three versions of the same system with three different levels of knowledge or capability.

The major problem in evaluating the MuRAL approach is the subjective judgment that is required from the perspective of each agent that a given plan step has been successful. In order to conclude objectively that a given plan has been successfully executed by a number of players, a special agent is required to test that condition reliably. However, such detection in an automated system in a realistic complex, dynamic, and uncertain domain is very difficult to perform objectively. For example, in the Soccer Server environment, determining whether a player currently possesses the ball or not can be done by a subjective test that checks whether the ball is in close proximity to the player. In general, however, there is no global monitoring entity that can provide such information. Therefore, testing of performance by an outside agent is just as difficult as evaluating performance from each agent's individual perspective. In MuRAL, agents check the *postconditions* in each plan step to test if that plan step has been successfully completed. If the time limit on a postcondition runs out or a failure occurs, then an agent concludes that the current plan has not been satisfied. If all agents who collaborate on a plan individually detect that the plan has been successfully applied, only then we consider that experiment a success.

To compare performance of the three versions of our system, we use the success rate as our evaluation metric. The *retrieval* and *search* versions of our system do not modify any knowledge or information that the system can use in the successive runs. Therefore, the experiments in these two modes are independent. In *learning* mode, each agent adds new information to its knowledge base via reinforcement, and, therefore, each experiment in this mode is dependent on all previous experiments.

3.6 Experiment Setup

During both training and application, we run controlled but situated experiments over a set of multiagent plans. A controlled experiment refers to a simulation run in which we select the type and the number of players, the initial layout of the players who are expected to execute the plan, and what plan to train on or apply. We then vary the number of opponent players for each plan, and place the opponent players randomly in and around the region where each plan is to be executed before we start each single run.

We conduct four main types of experiments summarized in *Table 3.1*. These refer to the four different versions of our system, one for the training mode and three for the application mode. For a given plan, we run each of these four main types of experiments for 1, 2, 3, 4, and 5 opponents. Since the number of players that are expected to execute a plan is determined by that plan, we only vary the number of opponents per experiment type.

To conduct all experiments, we automatically generate two sets of testing scenarios. The first set is the Training Set, and is composed of N randomly generated scenarios per plan per number of opponent players, and this test set is used only during training. The second set of scenarios is the Test Set, and is distinct from the Training Set, and is also composed of N randomly generated scenarios per plan per number of opponent players. Since we intend to compare the performance of learning, retrieval, and search, the Test Set is common to all three application modes.

	RL	Search	Retrieval	Case Retention
TRAINING	off	on	off	on
APPLICATION (Retrieval)	off	off	on	off
APPLICATION (Learning)	on	off	on	off
APPLICATION (Search)	off	on	off	off

Table 3.1: Experiment setup summary. The columns list the different modes we use to customize the behavior of each agent, and the rows list the type of experiment

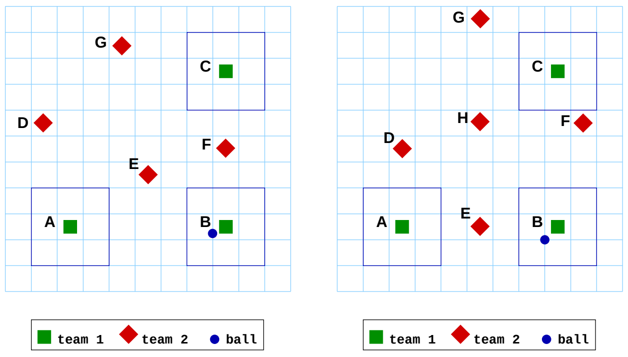
In training mode, opponent players do not move or execute any other behavior. The reason for using non-moving opponents is to allow MuRAL agents to retain as many solution as possible. If the opponent players are allowed to move, MuRAL agents may learn very little. The idea of training is, therefore, to retain action-level knowledge that can be *potentially* useful in less-controlled scenarios.

In the remaining three application mode experiments, the opponent players use a reactive ball interception strategy that is also used by team players that are executing a plan. If the opponents detect the ball is approaching towards them, they attempt to intercept the ball by moving and blocking the ball. Otherwise, they monitor the environment (See the Interceptball plan in *Section A.5*). [4] In both training and application experiments, we place the training agents in locations that will allow them to match the *internal conditions* of the plan initially. Later position and orientation of players is guided by the dynamics of each situation.

Two example test scenarios are shown in *Figure 3.15* for training on a plan that enables

[4]Training agents do not use the Interceptball plan per se. However, they do use the application knowledge that is used to implement the Interceptball plan, and this knowledge is used both by both MuRAL agents and opponent agents.

three agents in *team 1* to learn how to execute multiple passes to advance the ball in the field. Another group of agents, *team 2* acts as opponents. The training team, *team 1*, is identical in number and position in both scenarios. *Figure 3.15(a)* has 4 opponent *team 2* agents. *Figure 3.15(b)* has 5 opponent *team 2* agents, and these opponent agents are placed (randomly) at different locations compared to the first scenario.



(a) Training scenario 1 (b) Training scenario 2

Figure 3.15: Two example testing scenario for training on a threeway-pass plan

3.6.1 Training

The first type of experiment we run involves training, where agents try to find sequences of actions to implement the goals of each step in a given plan. Hence, the naïve RL and retrieval capabilities are inactive for training agents (*Table 3.1*). The goal is to collect application knowledge that contributed to the eventual success of each given plan and make all of this knowledge available to each player during the application mode. Therefore, each agent stores the application knowledge it discovers for its role in each plan step, if the application of that knowledge was successful. This information is kept distinct from the output of all other training runs for the same plan. In addition, each agent stores information about whether the entire plan has succeeded from its perspective or not. If all training agents store information that a particular training instance was successful, then we can use the knowledge stored by each agent with some confidence that it may also be successful in future similar situations. Therefore, it is not until the application of the given plan ends that we know whether the cases stored by the training agents can be potentially useful for later experiment stages. After all training ends for a plan, we use the success status information stored by each individual agent for each training run to decide whether to merge the application knowledge stored during that run into a single plan for each role. Although strictly not a part of the training, plan merging is a necessary and critical postprocessing step we perform to collect all successful cases for each role in a single plan. In merging plans, we exclude duplicate cases.

3.6.2 Application (Retrieval)

In retrieval mode, only the case retrieval capability is active (*Table 3.1*). In this stage, agents do not do search to discover new solutions. We assume that the application knowledge collected during training has been merged into a single plan for each role in that plan. The goal of this experiment is to measure the average success of the system when it is only using knowledge from training. The important distinguishing factor in this experiment is that the success and failure counts of cases are not modified after each application. Therefore, each retrieval mode test is independent of all other retrieval test. As in training mode, each agent stores information about whether the entire plan was successfully executed using only training knowledge.

3.6.3 Application (Learning)

In learning mode, both naïve RL and retrieval capabilities are active (*Table 3.1*). As in retrieval mode, agents do not do search but instead retrieve knowledge from their casebases for each plan step and for the role that an agent assumes. In addition, after the application of each case to the current test situation, each agent modifies the success or the failure count of that case. Moreover, in this mode, an agent that takes on a certain role in the given plan accesses and modifies the same plan created by plan merging for that role. Therefore, each learning mode test is dependent on all previous learning tests. As in other experiments, each agent stores information about whether the plan succeeded or not. Since each agent learns independently, there is no merging of knowledge through post-processing as in training. Therefore, different agents will have different reinforcement values for the same plan and role.

3.6.4 Application (Search)

Finally, in search mode, only the search capability is active (*Table 3.1*). Similar to the retrieval experiment, it consists of independent tests, and it only saves information about whether all steps of a given plan have been successfully completed. The search mode tests form our evaluation baseline, since search is the lowest-level application capability we use. Since search is a natural part of training, by comparing the performance of search with that from retrieval and learning experiments, we can draw certain conclusions about whether learning cases and reinforcing the usefulness of each individual case will have an impact on the overall behavior of the system.

3.7 Test Plans

We designed four distinct plans with varying complexity to test the ideas we put forth in this dissertation. Their layout on the field is depicted in *Figures 3.16*, *3.17*, *3.18*, and *3.19*. The shaded regions in all of these figures exemplify the regions within which opponent players are randomly placed. These regions and others where players are expected to be in or move to have been drawn to scale.

- The first plan is called Centerpass, and its layout is given in *Figure 3.16*. In this plan, player with role A is expected to dribble the ball from region R1 to region R2. Simultaneously, player B is expected to run from region R3 to region R4. After both complete this first step, player with role A needs to pass the ball to the player that assumed role B.

- The second plan is called Givengo[5], and its layout is given in *Figure 3.17*. In this plan, player with role A in region R1 needs to pass the ball to player with role B in region

[5]"Givengo" is pronounced "give-n-go."

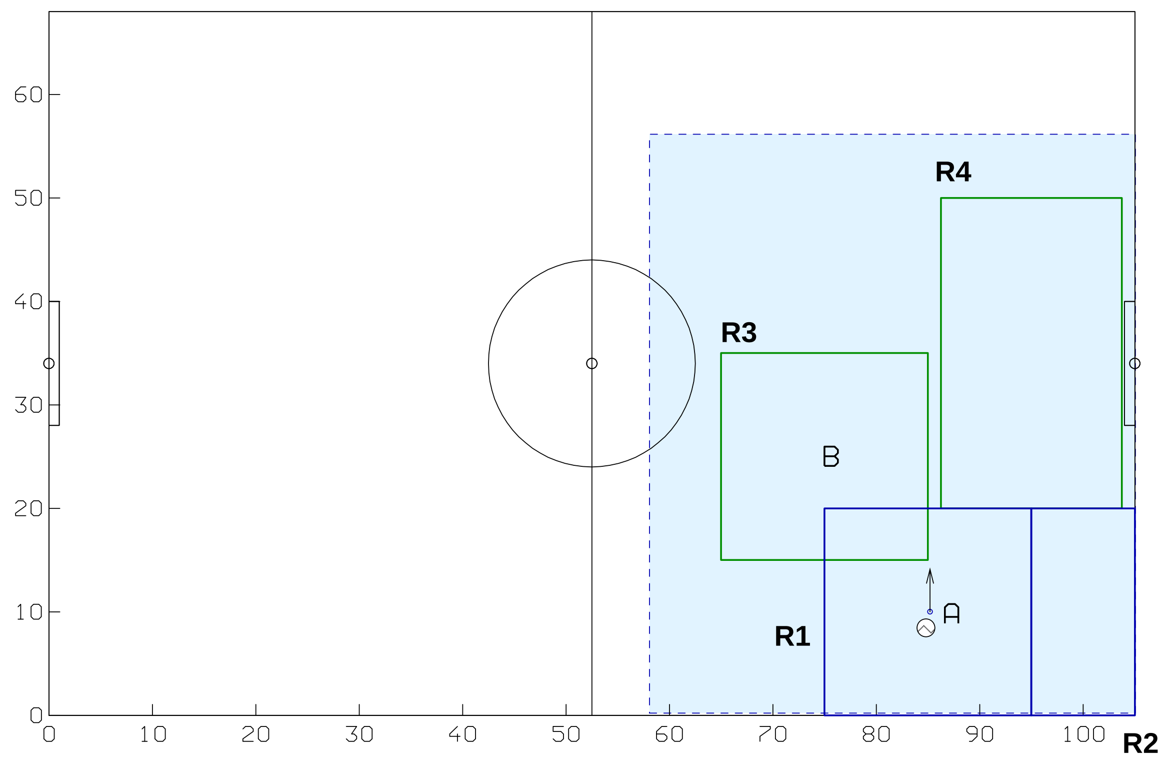


Figure 3.16: Centerpass plan scenario

R2. In the second step, A needs to run to region R3 while B keeps the ball. In the final step of the plan, B needs to pass the ball back to A in region R3.

- The third plan is called Singlepass. Its layout is given in *Figure 3.18*. As its name suggests, this plan involves one player passing the ball to another player. In this case, player with role A in region R1 needs to pass the ball to B in region R2. This plan has a single step.

- The fourth and last plan is called UEFA Fourwaypass [6] and is the most complicated plan among the four plans we designed with four steps and three roles.

  1. In the first step, B in region R2 needs to pass the ball to A in region R1.
  2. The second step involves A passing the ball it received from B to C in region R4.

[6]This plan has been borrowed from `http://www.uefa.com`. We use the name "Fourwaypass" to stress that the ball will be in four different locations by the end of this plan. This plan involves only three passes.

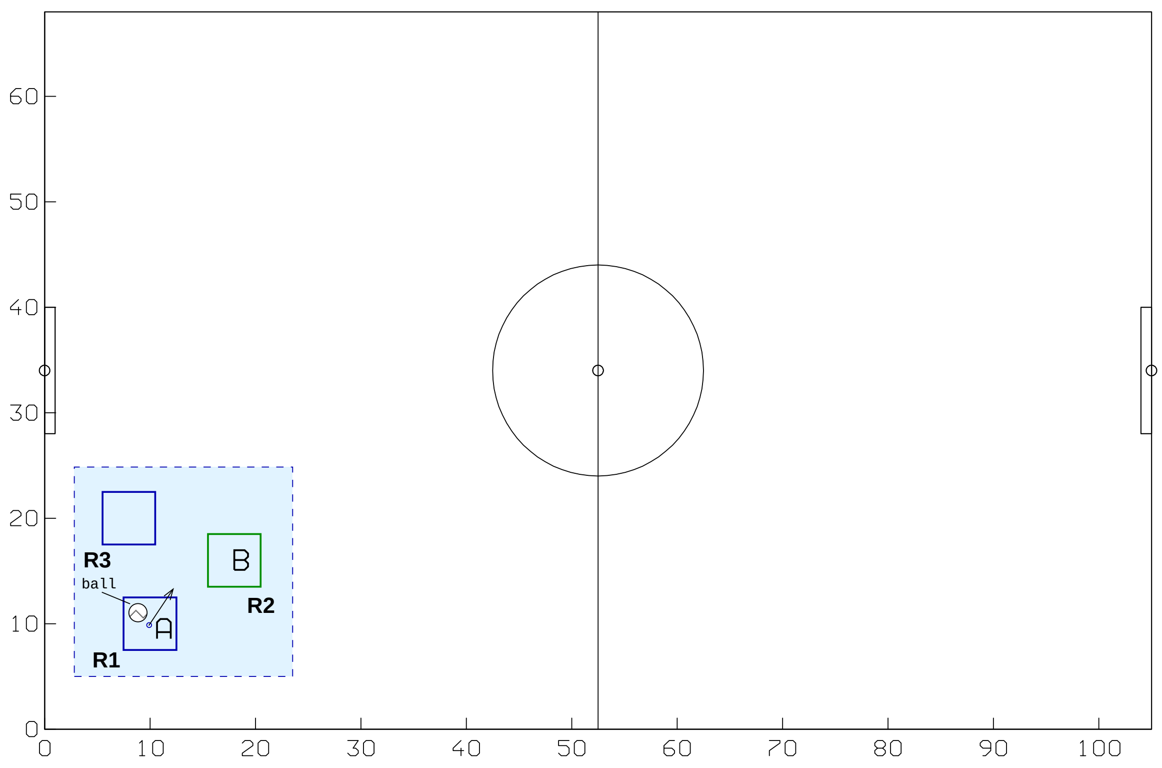


Figure 3.17: Givengo plan scenario

3. After passing the ball to C, A then needs to run to its second location in region R3.
4. Then, in the final step, A reaching region R3 triggers C to pass the ball back to A in region R3.

3.8 Related Work

The benefit of specifying high-level plans is to guide the solution process by constraining the learning problem for each plan. The idea of involving users to affect solutions has been acknowledged in the planning literature. For example, providing user guidance to a planning system in the form of strategic advice [Myers, 1996] or providing hierarchical plan sketches to be automatically completed by the system [Myers, 1997] are two such approaches. In this dissertation, we combine this general idea with learning so that autonomous and non-communicating agents can learn to implement reactive plans in

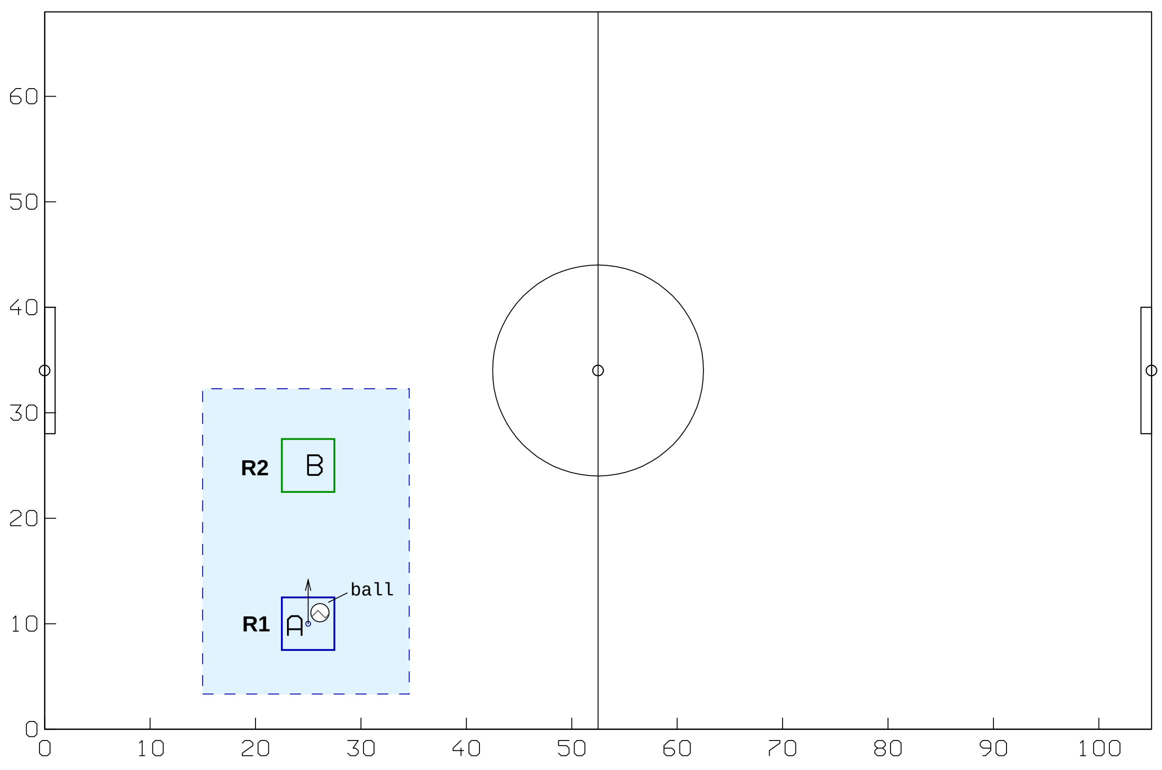


Figure 3.18: Singlepass plan scenario

complex, dynamic, and uncertain environments in a scalable manner. Agents in our approach have to be in continuous collaboration to be able to solve shared goals. Therefore, the MuRAL approach differs from cooperative distributed problem solving [Durfee et al., 1989], where problems solving agents need to coordinate to solve problems individual agents cannot solve.

In recent years, Reinforcement Learning (RL) has attracted a lot of attention from the multiagent learning community. Since the standard RL scheme is not applicable directly to complex multiagent domains, some researchers proposed modified approaches to multiagent learning in complex domains [Stone and Veloso, 1999; Matarić, 1991, 1996]. Stone's team-partitioned opaque-transition RL (TPOT-RL) approach [Stone and Veloso, 1999; Stone, 1998] (see also *Section 2.9.2*, page 52) does away with having to know the next state transition, and it uses *action-dependent features* to generalize the state space. Each agent learns how to act in its local environment, so this partitioning of the multiagent state space reduces the search. TPOT-RL evaluates each action based on the current state of the

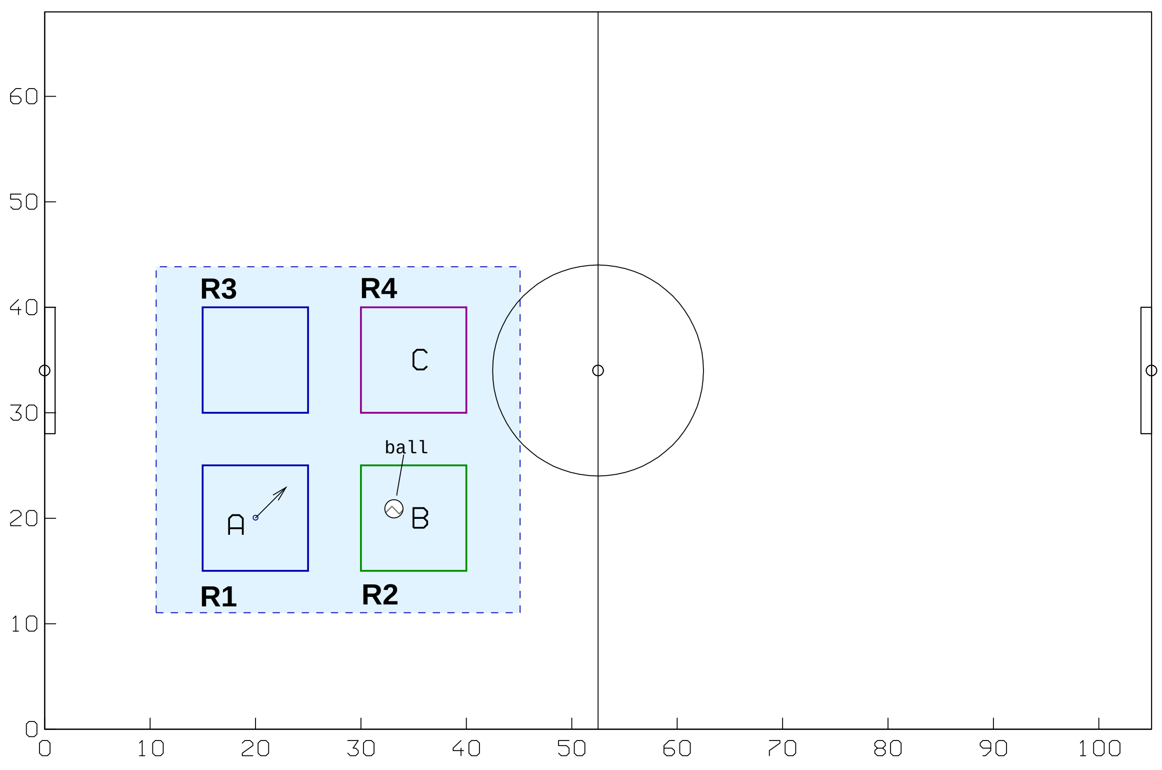


Figure 3.19: UEFA Fourwaypass plan scenario

world by classifying the potential short-term effects of that action using a learned decision tree, and it uses a reward function that provides feedback over a fixed period following the execution of an action. TPOT-RL deals with shifting concepts by not depending on future state transitions but depending only on the current state in which the action is taken. Despite these modifications, Stone later reported that the state aggregation approaches in TPOT-RL and in other techniques applied to a complex domain such as soccer "... are not well-suited for learning complex functions." [Stone and Sutton, 2001].

The work by Matarić reformulates the RL problem by using conditions and behaviors as opposed to the standard formulation that uses states and actions [Matarić, 1996; Mataric, 1994]. Conditions express a subset of the state space relevant for a given problem, and behaviors are more abstract than primitive actions. Therefore, a more general representation of the problem yields a reduced search space. Heterogenous reward functions allow the reinforcement of multiple concurrent goals and progress estimators provide additional domain-dependent reinforcement. Another advantage of using progress estimators is that they enable event-driven behavior termination so that more exploration can occur in cases where progress is not being made.

Mahadevan and Connell divided the overall RL task in a box-pushing domain into three domain-dependent manageable modules. Each module learned its own assigned task and a higher-level priority scheme activated one of the these modules during application [Mahadevan and Connell, 1992]. Similar to this work, research in scaling up RL to larger problems focuses on a single high-level task such as keepaway play in soccer (e.g., [Kuhlmann and Stone, 2003; Stone and Sutton, 2002, 2001; Kostiadis and Hu, 2001]). The number of total agents involved in these tasks is less than ten. Similar to Stone's layered learning approach, [Arseneau et al., 2000] addresses team-level learning where a single agent has the task of choosing between several high-level strategies and broadcasting it to its team members in order for them to fulfill their parts.

Similar to our approach, roles have been used to divide up the responsibilities among agents in multiagent systems. [Coradeschi and Karlsson, 1998] uses decision trees to describe the tasks of multiple roles hierarchically. In ISIS, team responsibilities are represented explicitly using an operator hierarchy. An operator in such a hierarchy expresses the joint activities of a soccer team, and it applies to the current mental state of the team, which is described by the agents' mutual beliefs about the state of the world. Coordination and communication handled using the teamwork model in ISIS called STEAM [Tambe, 1997b,a, 1996]. [Bowling et al., 2004] presents an approach for coordinating and adapting team behavior by way of symbolically represented team plans stored in a playbook. In this approach, a fixed-sized robot team has a set of plans called plays that may be appropriate for different opponents and situations. A play describes a coordinated sequence of team behavior, and a playbook stores all plays that a team can use. As the team applies plays from a playbook, it keeps track of which playbooks have been overall successful, so that it may be better able to select the most appropriate playbook as the team plays against other teams. Each play is made up of basic information and role information. The basic information describes the preconditions for applying a play and termination conditions. The role information describes the steps for executing a play. A timeout may also terminate a play. The team keeps track of whether a play was completed without a goal or not or whether it failed. Then it uses this information for adaptation. Each play has four roles, and the sequence of play/plan steps in each role specification is used to coordinate activities. Although similar to our representation approach in some respects, our work differs from this approach in one very fundamental way. [Bowling et al., 2004] uses a central controller that determines which team strategy should be used as opposed to our completely distributed approach. The representation language uses a simple language to express preconditions, termination conditions, and tasks for each role. In MuRAL, plan representation used describes roles from individual agent's perspective. The plan language in MuRAL is more complicated and more expressive. Another approach where responsibilities in a team are described as separate roles is described in [Simon Ch'ng, 1998b].

3.9 Summary

This chapter discussed the learning approach in MuRAL and presented the algorithms used. The prominent feature of this learning approach is the use of high-level symbolic plans to constrain search as agents learn how to operationalize plans using case-based learning and select plans and plan step implementation using a naïve reinforcement learning. In our implementation, agents are homogeneous, but each learns different knowledge from its own perspective depending on the roles it takes on over its lifetime. One clear advantage of our approach is that teams of agents can be trained on both existing and new plans any time. Once plans have a sufficient number of cases, agents can be deployed to apply them in full-fledged scenarios so that they can acquire reinforcement knowledge to learn to improve their selection of plans and plan implementations. This chapter focused on the theoretical approach in MuRAL. The next chapter focuses on the design and implementation issues of our system.
◇

Chapter 4

Implementation

This chapter describes the implementation of the MuRAL system developed for this dissertation. The agents we designed and implemented work in the Soccer Server simulator environment [Chen et al., 2002]. The implementation has been done entirely in C++ using object-oriented design principles and with the help of the Standard Template Library (STL). Although our implementation does not directly use any client code or library developed for the RoboCup simulator, [CMUnited source code] has served as a good example in developing basic soccer skills and inspired some of the related solutions.

4.1 Object Hierarchy

Figure 4.1 shows the main classes in the object class hierarchy of the MuRAL system. The class hierarchy is reminiscent of that in Java with an Object class at the top. The Object class is the parent class of almost all classes and provides a basic set of common functionality. In the figure, class names containing {#} refer to multiple classes whose names match the given pattern. For example, DashServerCommand and TurnServerCommand are some of the subclasses referred to by the class name pattern {#}ServerCommand, which is a subclass of ServerCommand.

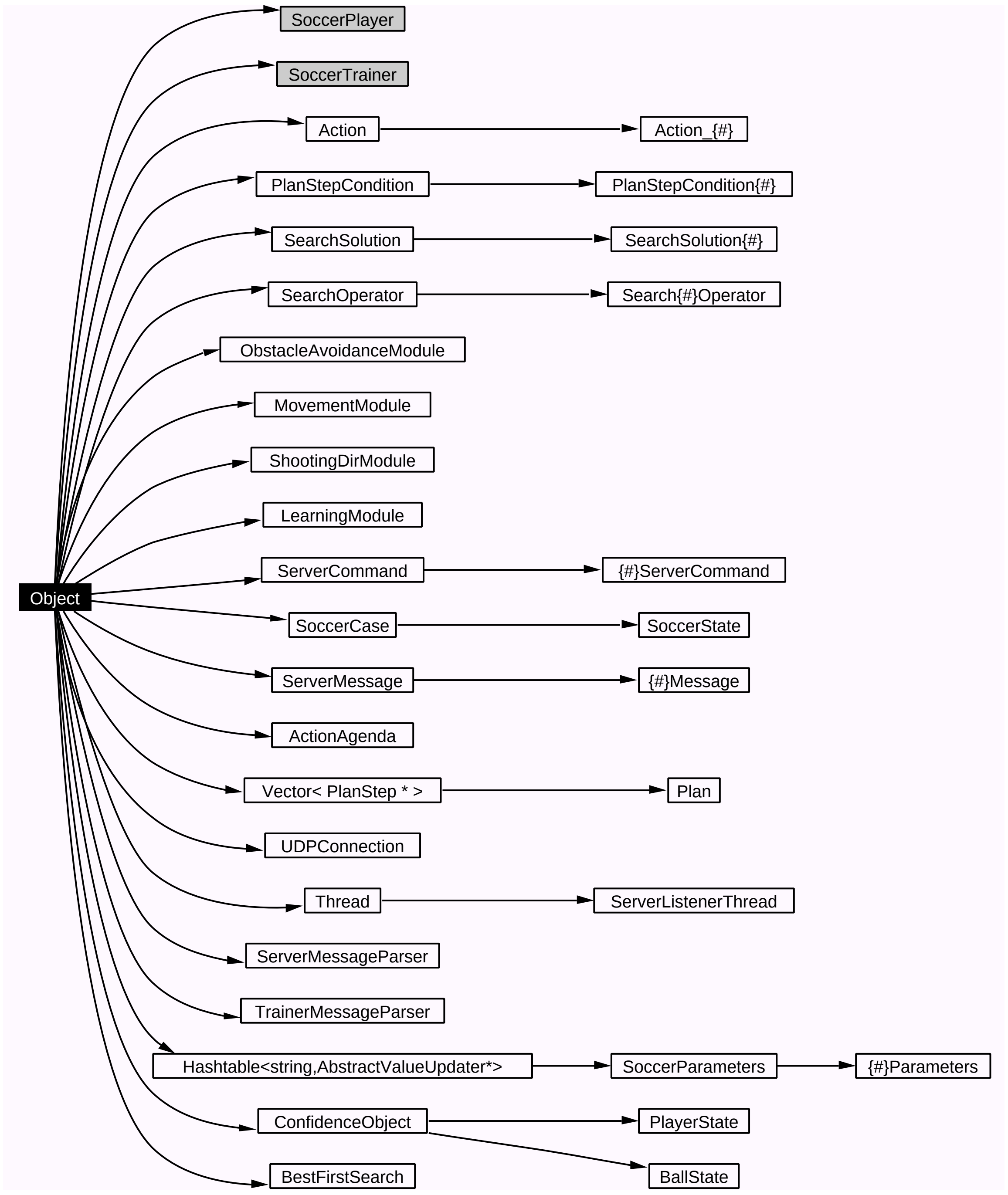


Figure 4.1: Object class hierarchy of the MuRAL implementation

Our implementation follows the highly parameterized design tradition of the Soccer Server. In addition to the parameter set defined by the Soccer Server, we define a large set of parameters that can be adjusted statically using an input file in order to customize tasks such as search, obstacle avoidance, ball interception, dribbling, and position triangulation to name a few. A MuRAL agent also processes the standard Soccer Server player configuration files. The SoccerParameters class and its subclasses implement this parameterization.

In MuRAL, there are two types of agents. *Player agents* train on plans and applying those plans in soccer games. We also use a single *trainer agent* to set up the scenarios in the simulator environment during both training and application. The trainer agent does not affect the reasoning of any of the agents. Player agents have a *strong* notion of agency while the trainer agent is simple and has a *weak* notion of agency [Wooldridge and Jennings, 1995]. Player agents are implemented by the SoccerPlayer class, and the trainer agent is implemented by the SoccerTrainer class. The Action class and its subclasses implement *reaction action modules* (RAMs). RAMs are executed on a stack-based agenda, implemented by the ActionAgenda class. RAMs also use functionality implemented in general-purpose modules (GPMs).

Reactive action modules (RAMs) depend on general-purpose modules (GPMs) we implemented, other RAMs, and the primitive actions provided by the Soccer Server. There are four general-purpose modules. The `Movement Module` (class MovementModule) implements navigation based on another GPM called the `Obstacle Avoidance Module` (class ObstacleAvoidanceModule) that implements obstacle avoidance. We implemented the obstacle avoidance scheme described in [Ulrich and Borenstein, 1998] [1]. The `Movement Module` builds an abstraction layer above the obstacle avoidance layer. It considers certain conditions to make navigation work more efficiently in noisy environments. For example, the `Movement Module` tries to avoid small turns when an agent is relatively far from its target location.

The `Shooting Direction Module` (class ShootingDirModule) enables player agents

[1] Also see [Ulrich and Borenstein, 2000].

to determine free directions for shooting the ball. Learning and training are implemented in the LearningModule class. The following is a description of the RAMs we implemented:

- DribbleBall: dribble the ball from the current location to a specified location on the field
- FollowBall: follow the position of the ball by turning the body and the neck of the player
- GotoBall: go to the ball and position the player according to a dynamically specified approach angle so that the player is in position to kick the ball in the intended direction
- GotoPosition: go to the specified target location
- GuardGoal: guard the goal by positioning the goalie based on the movement of the ball
- LocateBall: look for the ball when it is no longer visible
- InterceptBall: intercept an incoming ball and stop its movement
- PassBall: pass the ball to a specified player
- ScanField: scan the field to collect information about the world
- TurnBall: turn the ball around the player so that the ball can be kicked towards the desired direction

Figure 4.2 describes the dependence of RAMs and GPMs on other RAMs, GPMs, or primitive actions. RAMs are shown as ellipses, GPMs as octagons, and primitive actions as rectangles. The interdependence among RAMs strictly indicates contingencies. For example, InterceptBall depends on GotoBall in case the player misses the ball and has to run after it to collect it. Both InterceptBall and GotoBall depend on LocateBall to determine where the ball is when the player no longer sees it.

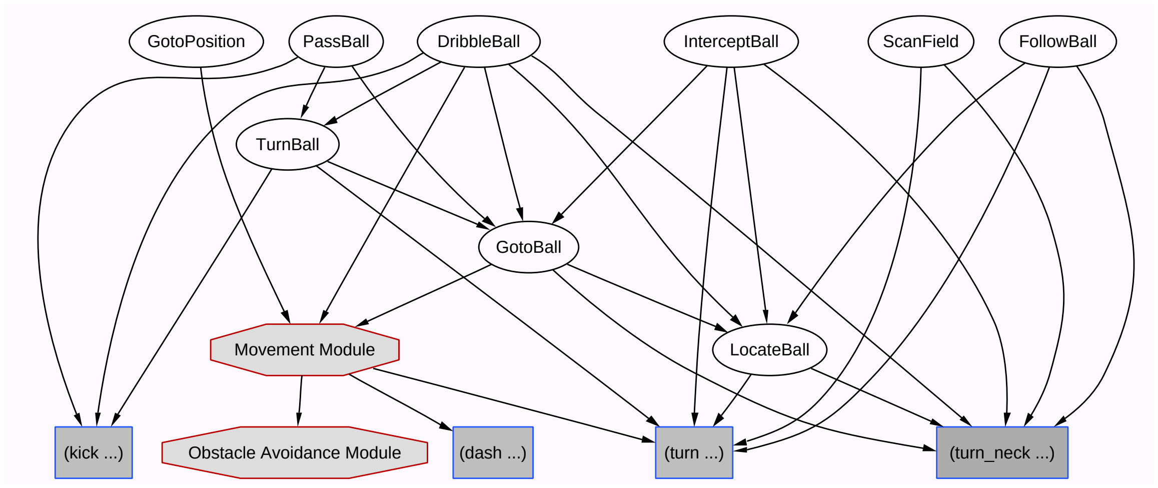


Figure 4.2: Reactive Action Module (RAM) dependencies on general-purpose modules (GPMs) and primitive Soccer Server actions

The communication subsystem that links agent programs the RoboCup soccer simulator is implemented by the following classes:

• The ServerMessage class and its subclasses represent the different types of messages that agents receive from the Soccer Server.

• The ServerMessageParser class implements parsing of server messages for player agents, and the TrainerMessageParser class implements message parsing for the trainer agent.

• The ServerListenerThread class enables player agents to collect messages sent by the simulator. [2] This class also automatically establishes synchronization with the simulator server (See *Section 4.3*).

• Agents send commands to the simulator using the subclasses of the ServerCommand class.

• The low-level communication capability in agents is implemented in by UDPConnection class, which provides UDP communication between clients and servers.

The search for action sequences for each plan step is implemented by the BestFirstSearch class. Search operators are subclasses of the SearchOperator. When an agent learns a

[2]Class ServerListenerThread has been implemented as a thread process but is used in non-threaded fashion in the current implementation.

new case, the search solution in that case gets translated to an action sequence which is represented at runtime by instances of the SearchSolution class. During application of each of these actions, the SearchSolution instance gets dynamically translated into a RAM, which is implemented in subclasses of the Action class. At runtime, each agent keeps track of the world state as it receives feedback from the simulator. An agent keeps the state of the ball in an instance of the BallState class and keeps the rest of the world state information, including its own state, in an instance of the PlayerState class.

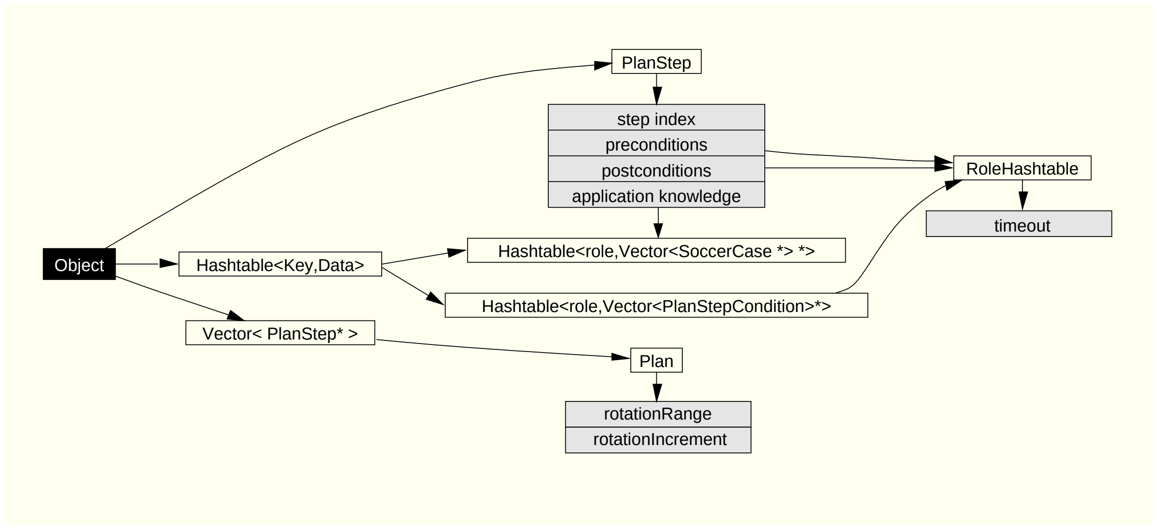


Figure 4.3: Runtime plan representation

Plans in MuRAL are represented by the Plan class. Cases are represented by the SoccerCase class. Individual conditions in preconditions and postconditions in plans are represented by the PlanStepCondition class. *Figure 4.3* is a more detailed representation of the data structures used to represent plans at runtime. The shaded boxes show the data kept by each data structure, as specified in the plan specification language (See *Section 4.2*). Each plan is a vector of plan steps. Preconditions and postconditions in each plan step are implemented as hashtables of list of conditions that are keyed by role names. Similarly, the application knowledge in each plan step is also a hashtable of a list of cases keyed by each role name in that plan step.

4.2 Plan Specification Language

In MuRAL, all plans are specified using a special language named MuRAL Plan Language (MuRAL-PL). The syntactical structure of this language is Lisp-like where parentheses are used to delimit expressions. *Figure 4.4* gives the general structure of MuRAL-PL.

```
(plan <symbol:plan-name>
   (rotation-limit <float[0..180]:max-rotation> <float[0..90]:rotation-increment>)
   (step <integer:plan-step-number>
      (precondition
         (timeout <integer:timeout-cycles>)
         (role <symbol:role-name> <integer:role-priority>
            ([not] <symbol:condition-name> <any:arg1> ... <any:argN>)
            ...
         )
         (role ...)
         ...
      )
      (postcondition
         (timeout <integer:timeout-cycles>)

         (role <symbol:role-name> -1
            (<condition> <arguments>)
         )
         (role ...)
         ...
      )
      (application-knowledge
         (case <symbol:plan-name> <symbol:role-name> <integer:plan-step-number>
            (gridunit <float:grid-unit>)
            (success <integer:success-count>)
            (failure <integer:failure-count>)
            (rotation <float:rotation>)
            (opponent-constellation (<integer:x-gridpos> <integer:y-gridpos>) ... (...))
            (action-sequence (<symbol:action-name> <any:arg1> ... <any:argN>) ... (...))
         )
         (case ...)
      )
   )
   (step ...)
   ...
)
```

Figure 4.4: MuRAL plan specification language

In *Figure 4.4*, MuRAL-PL keywords are shown in boldface. The meaning of each keyword and what arguments it takes is described below. Each argument is specified using the notation `<type[range]:name>`, where `type` is the data type of the argument (integer, float, symbol, etc.), `name` is a descriptive name of what that argument represents. In addition, for numeric values, a `range` specification may be given. A range is represented as `[a..b]` where `a` is the minimum value the argument can have and `b` is the maximum value it can have.

• **`plan`** starts a new plan and `plan-name` is the name of the plan. Each plan name in a given plan library must be unique.

• **`step`** defines each step in a plan. It is composed of preconditions, postconditions, and application knowledge (See *Figure 4.3*)

• The **`precondition`** section defines, for each role in a given plan step, the conditions that must be satisfied in order for that plan step to start. Similarly, the **`postcondition`** section describes the conditions that must be satisfied in order to terminate the current plan step. Each precondition and postcondition is described from the perspective of a given role in a given plan step.

• **`application-knowledge`** refers to the casebase associated with each plan step and can hold cases for each role in that plan step.

• **`timeout`** is the duration after which a non-satisfied precondition or postcondition is declared failed. This value is in terms of the simulator cycles. Its default value is 0, which means that the condition being checked for must either match or fail on the first try.

• **`rotation-limit`** defines how wide of an area an agent should check while checking for a match between the current situation and the current plan step. The first argument, `max-rotation` specifies the width of the area, in degrees, that an agent should check for a match to the current preconditions/postconditions. Half of the `max-rotation` value falls on one side of the current viewing direction of the agent and the other half falls on the opposite side of the agent's current viewing direction. The second argument, `rotation-increment` specifies the increments, in degrees, that the agent should use to test whether the current precondition/postcondition matches the current situation. *Figure 4.5* is an example of how the **`rotation-limit`** specification is used. In this case, `max-rotation` is 80 degrees and `rotation-increment` is 15 degrees. Checking for which rotation may match the current situation starts at the default 0-degree rotation of the current viewing angle and then moves to either side of the viewing direction in 15-degree increments. Since only two 15-degree increments can fit in a 40-degree sector on each side of the current viewing angle, the agent will check a total of five different rotations at 0, +15, +30, -15, and -30 degrees relative to the current viewing angle. The remaining 10-degree sectors on each edge of the 80-degree sector will not be checked since the next increment (±45) would fall outside the 80-degree sector.

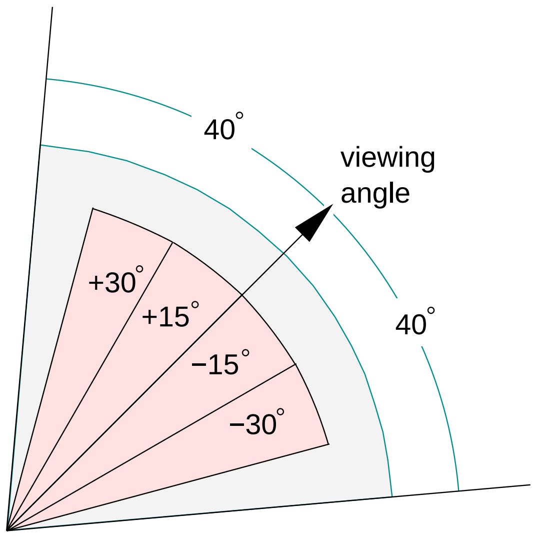


Figure 4.5: An example `rotation-limit` specification in plans with 80-degree maximum rotation and 15-degree rotation increments

• **`role`** defines each role that must contribute to executing or learning about executing a given plan step. A role is defined by the name of the role (`role-name`), priority of the role (`role-priority`), and a list of conditions. A condition is defined by its name (`condition-name`) and a list of arguments that can be of any valid type. A condition may be negated with the `not` keyword. The priority value in a role specification is used to order which matches to check for first.

• **`case`** defines a new case. Each case is associated with a plan (`plan-name`), a role (`role-name`), and a plan step (`plan-step-number`) so that each can be uniquely identified. At the least, it important to distinguish cases for each role in the current plan step. The value of `plan-name` must be the same name as that specified for the current plan, the value of `role-name` must match one of the roles specified in the current plan step, and the value of `plan-step-number` must be identical to the current plan step. The remaining values in a **`case`** specification can be specified in any order. The **`gridunit`** field specifies the unit distance of each grid square. All coordinate specifications (See, for example, `opponent-constellation` below) in a case are in grid coordinates. The fields **`failure`** and **`success`** specify the number of times the current case was either successfully used or failed. The **`rotation`** value specifies by how many degrees the opponent team scenario stored in this case was offset from the viewing angle of the agent. The **`opponent-constellation`** field stores the opponent constellation that was relevant to the current step of the plan and is made up of a list of (x, y) grid locations. The ***action-sequence*** field stores the list of actions that must be executed in the order specified. Each action is specified by a name that refers to that action and a list of arguments of any valid type. At the time a case is created, **`rotation`** value is used to *normalize* the scenario by rotating the coordinate values in the opponent constellation and in the action sequence specification so that all coordinate values are with respect to the viewing angle of the agent. During matching and execution, an agent starts with this *normalized* case representation, and, depending on the angle offset from the current viewing direction at which it finds a match to the current scenario, it rotates the appropriate case representations to match them to the current scenario. During application, the grid world coordinates in an action sequence specification are converted to actual world coordinates.

The next two subsections list the plan condition primitives and action types that are available for writing plans.

4.2.1 Plan Condition Primitives

- `has-ball`

  Checks whether the specified player has possession of the ball

- `in-rectangle-rel` *`x1 y1 x2 y2 x3 y3 x4 y4`*

  `in-rectangle-abs` *`x1 y1 x2 y2 x3 y3 x4 y4`*

  Checks whether the specified player is in the given region, defined by its four corners *`(x1,y1)`*, *`(x2, y2)`*, *`(x3, y3)`*, and *`(x4, y4)`*.

- `ready-to-receive-pass`

Always returns true. This condition causes a player to go into ball interception mode.

- `play-mode` *`mode`*

  Checks whether the current play mode is *`mode`*.

- `true`, `false`

  These conditions cause automatic satisfaction or failure of a condition.

- `uniform-no` *`num`*

  Checks whether the uniform number of the specified player is *`num`*.

4.2.2 Case Action Primitives

- `goto-area-rel` *`x1 y1 x2 y2 x3 y3 x4 y4`*

  `goto-area-abs` *`x1 y1 x2 y2 x3 y3 x4 y4`*

  Moves the player to the specified relative or global rectangular area whose four corners are given.

- `goto-position-re` *`x1 y1`*

  `goto-position-abs` *`x1 y1`*

  Moves the player to the specified relative or global point.

- `dribble-ball-rel` *`x1 y1 .. xn yn`*

  `dribble-ball-abs` *`x1 y1 .. xn yn`*

  The player dribbles the ball to a set of relative or global waypoints, [*`(x1, y1)`* .. *`(xn, yn)`*].

- `intercept-ball`

  The player activates ball interception.

- `pass-ball` *`R`*

  The player passes the ball to another player that has been assigned the tasks of role *`R`*.

- `pass-ball-to-area-rel` *`x1 y1 x2 y2 x3 y3 x4 y4`*

  `pass-ball-to-area-rel` *`x1 y1 x2 y2 x3 y3 x4 y4`*

  The player passes the ball to a relative or global area defined by its four corners *`(x1,y1)`*, *`(x2, y2)`*, *`(x3, y3)`*, and *`(x4, y4)`*.

- `scan-field` [*`use_body`*`|`*`no_use_body`*]

  The player scans the field to collect information about the game either by turning its body (*`use_body`*) and observing relevant changes or by only turning its neck (*`no_use_body`*).

- `goto-ball`

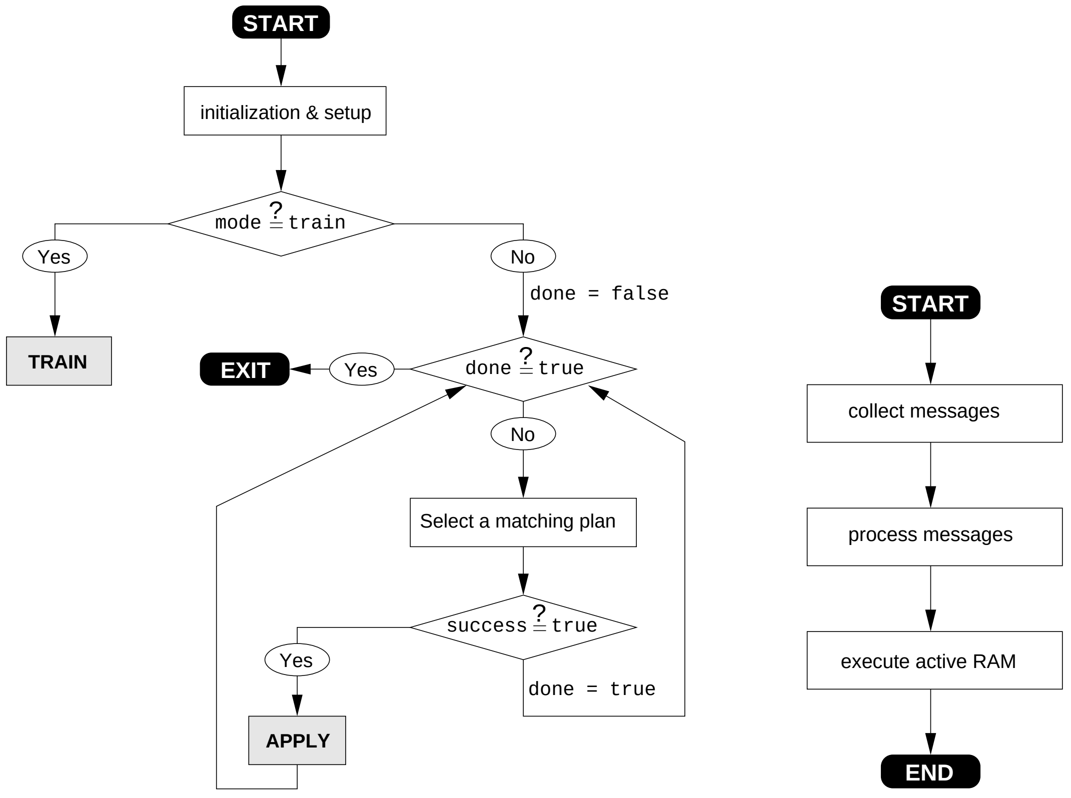
  Moves the player to the ball such that the player can kick the ball if needed.

4.3 Agent Operation

In the simulated soccer environment, an agent works as a client that connects to a server (simulator) through a UDP socket connection. In general, the agents do not communicate among themselves except through the simulator. [3] *Figure 4.6* shows the main operation cycle and the basic process cycle of a MuRAL agent. The main operation cycle in *Figure 4.6(a)* summarizes the overall behavior of an agent, and the basic process cycle in *Figure 4.6(b)* shows the critical steps an agent takes in each cycle of its operation.

The main operation cycle (*Figure 4.6(a)*) starts with the initialization and setup of the agent, which includes connecting to the simulator. Since a MuRAL agent can operate in one of two modes (training, application), the agent checks whether it is in *training mode*. If so, it executes its training algorithm (*Section 3.4.2*). If not, it means that the agent is in *application mode*. The application mode is a loop that allows an agent to successively select a plan and execute it in the actual environment. If the agent can find a matching plan, it executes its plan application algorithm (*Section 3.4.4*, page 115). If it cannot find any more matching plans to apply, the application loop ends and the agent halts.

[3]For the practical reason of shutting down each experiment trial, our agents send their process ids to the Trainer agent.



(a) Main operation cycle (b) Basic process cycle

Figure 4.6: Main operation cycle and basic process cycle of a player agent

In every simulation cycle during this main operation cycle, an agent follows a basic process cycle shown in *Figure 4.6(b)* (Also see *Figure 3.3*). First, the agent collects all messages sent by the simulator. Second, it processes these messages by parsing them and updating its world model accordingly. In this step, one of the critical operations an agent carries out in the RoboCup simulated soccer environment is position triangulation, because it is critical for a player to know where it is and how it is oriented on the field. The agent uses the fixed markers within its visual range to triangulate itself. The triangulation algorithm used by MuRAL agents is described in *Appendix D*. To update its world model based on the visual feedback it receives from the simulator, an agent computes the position and the velocity of the ball and also resolves its previous observation of other agents with respect to the new visual update it received. As we mentioned in *Section 3.3.3*, we use confidence values to express the dependability of information about currently unseen objects. This player resolution has three steps:

1. Computation of the orientation of all players in the visual update
2. Incorporation of newly seen players into existing memory of previously seen objects to try to deduce missing information about currently unseen objects and update their confidence values
3. Removal of low-confidence objects

Knowing the velocity of the ball is critical in intercepting the ball, and resolving visual updates allows an agent to build a continuous representation of the world state instead of one that is episodic, where each new visual update would overwrite all relevant parts of the world state. When an agent turns its body or neck, the objects that were previously visible may no longer be visible. However, it is advantageous to keep memory of previously seen but currently unseen objects. It is difficult to estimate the velocity of moving objects in the simulator environment from position information in subsequent cycles alone due to the noise in the visual updates about object movements. On the other hand, the simulator provides approximate information on each moving object so that its velocity can be more reliably estimated. *Appendix E* describes how we implemented this velocity estimation and gives the derivation of the velocity formulas, which are based on how the simulator computes related values it reports to players [Chen et al., 2002].

The synchronization of client programs with the server is another critical aspect of implementing agents in the RoboCup simulator environment. The simulator advances to the next cycle at regular intervals and without waiting for any of the clients. Therefore, each client has a limited time to decide what to do next. It is also critical at what time during a simulator cycle an agent sends a command in order for that command to take effect before the next simulation cycle starts. Therefore, knowing when a new simulator cycle starts is important to maximize the amount of time an agent has to respond to the current situation and update its world model in synchronization with the server. To synchronize agents, we use a procedure that takes advantage of a feature of the simulator. This synchronization procedure is described in *Appendix C*.

The third main step in the basic process cycle (*Figure 4.6(b)*) involves executing the currently active *reactive action module* (RAM). Each MuRAL agent has a stack-based *action agenda* (implemented in class ActionAgenda) that stores and executes RAMs that are activated by the current plan step. The `action-sequence` field in a case lists the actions an agent needs to execute. These actions have corresponding RAMs. Each corresponding RAM is then pushed onto the *action agenda* in the order it appears in the `action-sequence` field, and the topmost RAM is executed. When contingencies occur, a RAM may invoke other RAMs as shown in *Figure 4.2*.

◇

Chapter 5

Experimental Results and Discussion

In this chapter, we present and interpret the results of our experiments in light of our main hypothesis that naïve RL can improve the performance over training knowledge in multiagent plan application in complex, dynamic, and uncertain environments. Our system started with a set of handwritten plans in skeletal form. Each plan specified what each role in each step needed to accomplish in order to achieve the goal of that plan. These plans contained only the necessary information to describe *what* each agent that took on a given role in that plan had to do, but they did not have any application-specific knowledge for *how* to carry out each plan step. The goal of training was to acquire this application knowledge from a large number of situated training scenarios we referred to as the Training Set in *Section 3.6*. Following training, we used a separate Test Set to measure the performance of the *learning* version of our system against two non-learning versions of the same system. Both of the non-learning versions of our system did not modify the knowledge we obtained from training. The *retrieval* version only retrieved cases and applied them to test situations. The *search* version did not use any of the knowledge collected during training and instead used search dynamically to find solutions for each step of the plan.

5.1 Introduction

Our agents build solutions to different soccer problems in terms of sequences of actions selected and instantiated from a small set of reactive action modules (RAMs) (Also see *Section 3.3.5*). Some of these RAMs we implemented for simulated soccer require numeric parameters and others do not. RAMs that implement ball interception and passing, for example, do not take any numeric parameters in plans. What this means is that, if the search on the postconditions of a plan step results in one of these parameterless actions, the agent will save only one case representing all possible solutions during training, since all cases will be identical by default. RAMs for ball dribbling, on the other hand, require real-valued parameters. Therefore, such actions can be instantiated with a wide range of numeric arguments for each dynamic situation.

Since the game of soccer revolves around passing the ball among players, the most interesting behavior also occurs when dribbling and passing are combined. Since each case retained during training is matched using conditions external to the training agent group, opponent player constellations on the field contribute significantly to the improvement of the coverage of the domain by the agent casebases.

Next, we present the training experiment results followed by the application experiment results.

5.2 Training Experiment Results

Table 5.1 gives the number of training trials we used for each plan in each of the five opponent scenarios (See *Section 3.6*). Even though we used different number of trials in some of the experiments, the retrieval and learning experiments started with the identical copies of the operationalized plan that we obtained after merging the knowledge from all five opponent trials in each case. That is, the knowledge used in both retrieval and learning experiments came from the same training session for each given plan. Since we only merge the knowledge from trials that were successful for the entire plan, the actual number of trials that contributed knowledge for the application phase is less than the number of trials shown in *Table 5.1*. Even though opponents do not move during training, not all training trials resulted in success.

	1	2	3	4	5
Centerpass	1000	1000	1000	1000	1000
Givengo	1000	1000	1000	1000	1000
Singlepass	1000	1000	1000	1000	1000
UEFA Fourwaypass	1000	1000	1000	550	630

Table 5.1: Number of trials used during training for each plan for all 5 opponent scenarios

The application knowledge that our system acquired during training is summarized in *Tables 5.2*, *5.3*, *5.4*, and *5.5*. For each of the four plans we designed for this dissertation, these tables list how many cases were retained for each role in each plan step.

The source code of the four plans (Centerpass, Givengo, Singlepass, and UEFA Fourwaypass) we used as input to training in this dissertation are given in *Appendices A.1*, *A.2*, *A.3*, and *A.4*. For an example of how a plan is modified after training and learning, see an instantiation of the Givengo plan in *Appendix B*.

As *Table 5.2* indicates, the two agents in the Centerpass plan only learned one case each for both steps of the plan. This has to do with the limitations of the plan that makes agent A dribble the ball close to the goal line of the opponent team before passing the ball to agent B. When A finishes step 1, it is always looking straight ahead and that prevents A from seeing the opponents to its left. As a result of this limitation, the cases retained are the minimum set of actions needed to accomplish the Centerpass plan. Since there is no variation in the cases, there is minimal coverage of the domain.

Step#	Role	#Cases
1	A	1
	B	1
2	A	1
	B	1

Table 5.2: Number of cases acquired for each role in each step of the Centerpass plan during training

Table 5.3 lists the number of cases retained during training for the Givengo plan. In step 1, 90 cases were retained for role A to accomplish the pass from role A to role B. In step 2, 770 cases were retained for role A to implement the goal of moving A to its second location. In step 3, role B learned 9 cases for passing the ball back to A. Since role B did not have a critical task to perform in step 2, it had nothing to learn.

Step#	Role	#Cases
1	A	90
	B	1
2	A	770
	B	0
3	A	1
	B	9

Table 5.3: Number of cases acquired for each role in each step of the Givengo plan during training

Since the Singlepass plan involves the execution of a pass between two players, the critical work from the perspective of our approach and implementation rests with role A, which is responsible for passing the ball. Role A has to intercept the ball, and, therefore, only the ball interception action is sufficient for it to satisfy its role in the plan; hence the single case for B. To accomplish the pass, on the other hand, training generated 210 unique cases for role A (*Table 5.4*).

Step#	Role	#Cases
1	A	210
	B	1

Table 5.4: Number of cases acquired for each role in each step of the Singlepass plan during training

Table 5.5 lists how many cases were retained during training for the UEFA Fourwaypass plan. The critical roles for case coverage in this plan are role B in step 1, role A in step 2, role A in step 3, and finally role C in step 4. The remaining roles are associated with ball interception. Role B in step 1 has 41 cases, role A in step 2 has 103 cases, role A in step 3 has 581, and, finally, role C in step 4 has 2 cases. Role C in step 1, role B in step 2, roles B and C in step 3, and role B in step 4 did not have critical tasks; therefore, training agents learned no cases for those roles. When a role does not have any critical plan-based tasks to perform in a given plan step, our system automatically assigns a default behavior to the corresponding agent. This default behavior causes an agent to continually monitor its environment and watch for an incoming ball until it needs to perform a plan-based task in another plan step.

Step#	Role	#Cases
1	A	1
	B	41
	C	0
2	A	103
	B	0
	C	1
3	A	581
	B	0
	C	0
4	A	1
	B	0
	C	2

Table 5.5: Number of cases acquired for each role in each step of the UEFA Fourwaypass plan during training

5.3 Application Experiment Results

Following training, we ran three types of application experiments over the same Test Set to compare the performance of our learning approach to the retrieval and search versions of the system. This section presents the results for all three types experiments for each of the four plans we used in this dissertation (See *Section 3.7*). For each plan, we present five sets of summaries in tabular form to describe the behavior of our system in *learning*, *retrieval*, and *search* modes:

1. The graph of the running success rate of the plan during learning for all five opponent scenarios. The x-axis of the graph is the number of experiments run, and the y-axis is the average success up to that point in testing.

2. The success rate of the plan in the three application experiments for five different scenarios with 1 to 5 opponent players. The size of the Test Set was 1000 for both the

retrieval and search experiments and 2000 for the learning experiments in all four plans. The test trials that did not run properly were considered neither as successes nor failures.[1] Hence all graphs have less than 2000 points.

For the retrieval and search experiments, we compute the success rate using $\#Successes/(\#Successes + \#Failures)$. For the learning experiments, we use the average of the last 100 test trials, since the learning tests are dependent on all previous trials and hence their success rate is cumulative over the number of experiments run so far.

3. The mean and standard deviation of the last 100 values from each specific learning experiment to show that the success rate converges.

4. The paired t-test analysis for statistical significance of the overall difference among the learning, retrieval, and search performance of each plan. This table lists four columns: (1) The type of the comparison (RL vs. Retrieval, RL vs. Search, Retrieval vs. Search). (2) The probability value (p-value) for accepting the Null Hypothesis that the distributions being compared are equal. (3) The t-statistic (t-value). (4) Finally, the best confidence level for rejecting the Null Hypothesis [Cohen, 1995; Mendenhall and Sincich, 1992].

5. Finally, the paired t-test analysis for each plan across the three application elements but for each opponent scenario separately. For these individual opponent scenario comparisons, we correspond each unique test trial to obtain the results. This table lists six columns: (1) The number of opponents used in the experiments being compared. (2) Number of individual test trials that ran properly under all three techniques so that we can do paired comparisons using the t-test. The value given in this column is the maximum number of paired individual test trials. (3) The type of the comparison (See item 4 above). Columns (4) and (5) contain the same information

[1]During testing, we observed that some test trials were not being properly set up to run by our testing procedure. This caused the given scenario never to be tried. Since we consider success based on positive evidence that all agents succeeded in executing the given plan in its entirety, such improper test instances will, in general, look as failures. Since we consider them as neither successes nor failures, they do not affect the results we report in this section.

mentioned in item 4 above. (5) Whether the Null Hypothesis that the distributions being compared are equal can be rejected with at least 95% confidence.

We use the *paired t-test* analysis, since we used different number of opponent players to test each plan's performance across the three main experiment types; but, for each opponent scenario, we used the same test set across the three experiments.

5.3.1 Centerpass Plan

Figure 5.1 shows the success rate of the Centerpass plan with learning for each of the five opponent scenarios. As we would intuitively expect, the performance of the two collaborating agents in the Centerpass plan is best with 1 opponent player. The learning performance then decreases as we add more opponents to the test scenarios. This is indeed true, as we will see, of the performance of the remaining three plans.

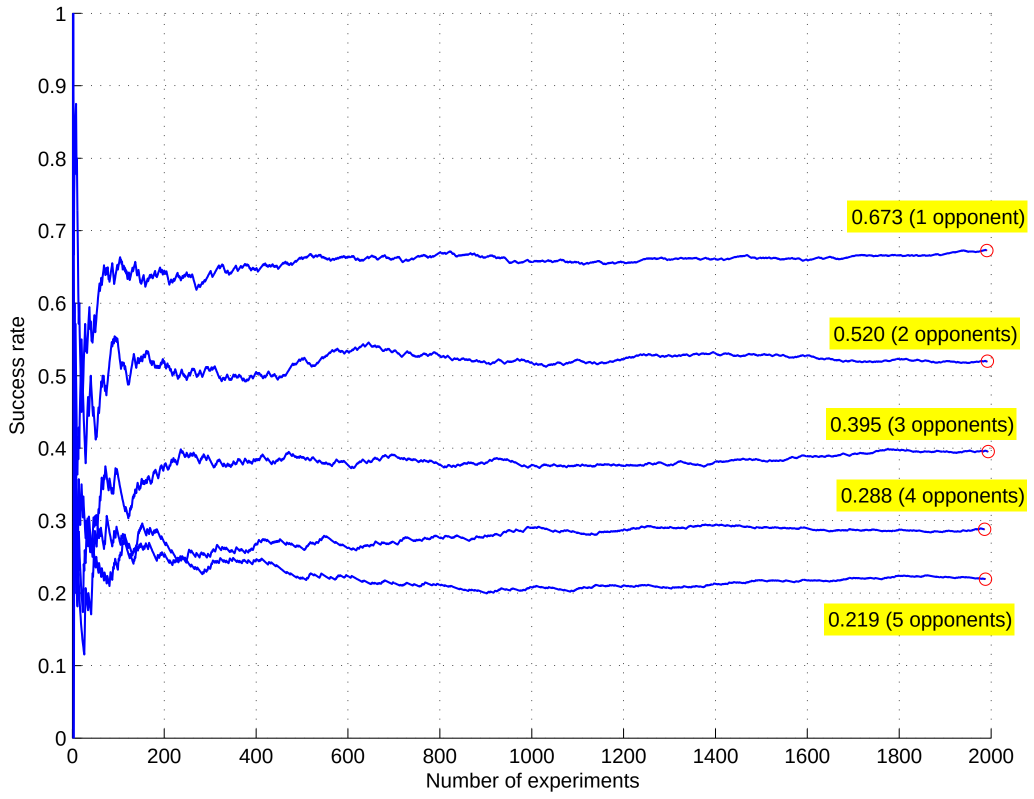


Figure 5.1: Learning experiments using the Centerpass plan. The last success rate value and the number of opponent players used is indicated on each graph

Table 5.6 is the summary of all three experiments for the five opponent scenarios. The rows of this table represent the type of the experiment, and the columns represent the average success rate of each experiment. Since each single Retrieval and Search test is independent of any other, the Retrieval and Search values in *Table 5.6* represent the percent success rate. Since each RL experiment trial is dependent on previous trials, we use the last 100 values to obtain an average success rate value as we described earlier.

	1	2	3	4	5
RL	0.671	0.517	0.395	0.286	0.221
Retrieval	0.626	0.509	0.347	0.289	0.216
Search	0.042	0.043	0.018	0.013	0.018

Table 5.6: Success rates of Centerpass plan experiments with [1 .. 5] opponents in RL, Retrieval and Search modes

Table 5.7 shows the mean and standard deviation of the success rate of the last 100 learning trials using the Centerpass plan. As the standard deviation values indicate, the learning success rate converges in all cases.

	1	2	3	4	5
mean	0.671	0.517	0.395	0.286	0.221
std. dev.	0.002	0.004	0.001	0.001	0.001

Table 5.7: The mean and standard deviation of the last 100 values from the Centerpass plan RL experiment

Table 5.8 lists the results of the overall paired t-test analysis. It compares the performance of the three techniques (RL, Retrieval, and Search) in the context of the Centerpass plan. From *Table 5.6* we can see that the success rates of the RL and Retrieval experiments are very similar. On the other hand, there is a marked difference between the success rates of RL and Retrieval performance compared to the Search performance. These similarities and differences are revealed in *Table 5.8*. We find that there is no statistically significant difference between the RL and Retrieval performance at the 95% confidence level but that there is still an 85% confidence.

On the other hand, we find highly statistically significant difference between RL and Search with 99% confidence. The same is true when we compare Retrieval to Search. The relative differences or similarities between these results are expected, since the Centerpass plan did not have many cases after training. The reason for the Search performance to be as low as it is has to do with how search works in general. When an agent reasons using search, it must "freeze" the world it sees and try to find a solution. However, doing so does not take into account the dynamics of the environment. Search, in theory, can attempt to predict how other agents may behave within a limited time window into the future, however, such predictions cannot guarantee that is how the behavior will be of the world external to the agent. Especially when we take into account the combinatorial growth of multiagent search spaces (*Section 3.3.5*), it becomes clear that search alone is not sufficient to solve very complicated multiagent problems in dynamic environments.

comparison	p-value	t-value	confidence
RL vs Retrieval	0.120	1.974	85.0%
RL vs Search	0.006	5.212	99.0%
Retrieval vs Search	0.006	5.371	99.0%

Table 5.8: Paired t-test results for comparing Centerpass plan experiments run in RL, Retrieval, and Search modes

Table 5.9 gives the results of the individual paired t-tests for the three application techniques for each of the five opponent scenarios using the Centerpass plan. In this table, we compare the performance of learning, retrieval, and search within the context of each specific opponent scenario instead of comparing the overall performance of these three techniques with the Centerpass plan as in *Table 5.8*. Consistent with the overall RL vs. Retrieval result in *Table 5.8*, we find that RL performs no better than Retrieval in all five distinct opponent setups. However, both RL and Retrieval always perform significantly better than Search.

5.3.2 Givengo Plan

As in the case of the Centerpass plan, the performance of the Givengo plan decreases as we increase the number of opponent players during the learning experiment. *Figure 5.2* gives the performance of the five individual opponent instantiations of the learning experiment using the Givengo plan.

#opponents	#trials	comparison	p-value	t-value	confidence
1	979	RL vs Retrieval	0.071	1.805	–
		RL vs Search	0.000	37.546	95.0%
		Retrieval vs Search	0.000	34.259	95.0%
2	988	RL vs Retrieval	0.574	0.562	–
		RL vs Search	0.000	28.107	95.0%
		Retrieval vs Search	0.000	27.341	95.0%
3	980	RL vs Retrieval	0.106	1.620	–
		RL vs Search	0.000	22.634	95.0%
		Retrieval vs Search	0.000	21.260	95.0%
4	996	RL vs Retrieval	0.908	0.115	–
		RL vs Search	0.000	18.708	95.0%
		Retrieval vs Search	0.000	18.525	95.0%
5	988	RL vs Retrieval	0.655	-0.447	–
		RL vs Search	0.000	13.998	95.0%
		Retrieval vs Search	0.000	14.273	95.0%

Table 5.9: Paired t-test results for comparing the performance of RL, Retrieval, and Search modes in each opponent experiment in the Centerpass plan

As summarized in *Table 5.10*, compared to the Centerpass case, Search in the context of the Givengo plan actually performed well compared to RL and Retrieval. While both the RL and Retrieval performance ranged roughly between 75% and 20%, the Search performance ranged roughly between 67% and 20%.

	1	2	3	4	5
RL	0.758	0.554	0.456	0.325	0.243
Retrieval	0.741	0.544	0.382	0.306	0.204
Search	0.673	0.491	0.370	0.258	0.196

Table 5.10: Success rates of Givengo plan experiments with $[1 \mathinner{..} 5]$ opponents in RL, Retrieval and Search modes

Table 5.11 gives the mean and the standard deviation of the success rate of the last 100 learning trials in all five opponent scenarios for the Givengo plan. This table demonstrates that the success rate of learning has converged.

The paired t-test analysis summarized in *Table 5.12* shows that there are statistical differences among RL, Retrieval, and Search performances but that there is no statistically significant difference between RL and Retrieval. Numerically, RL performed the best,

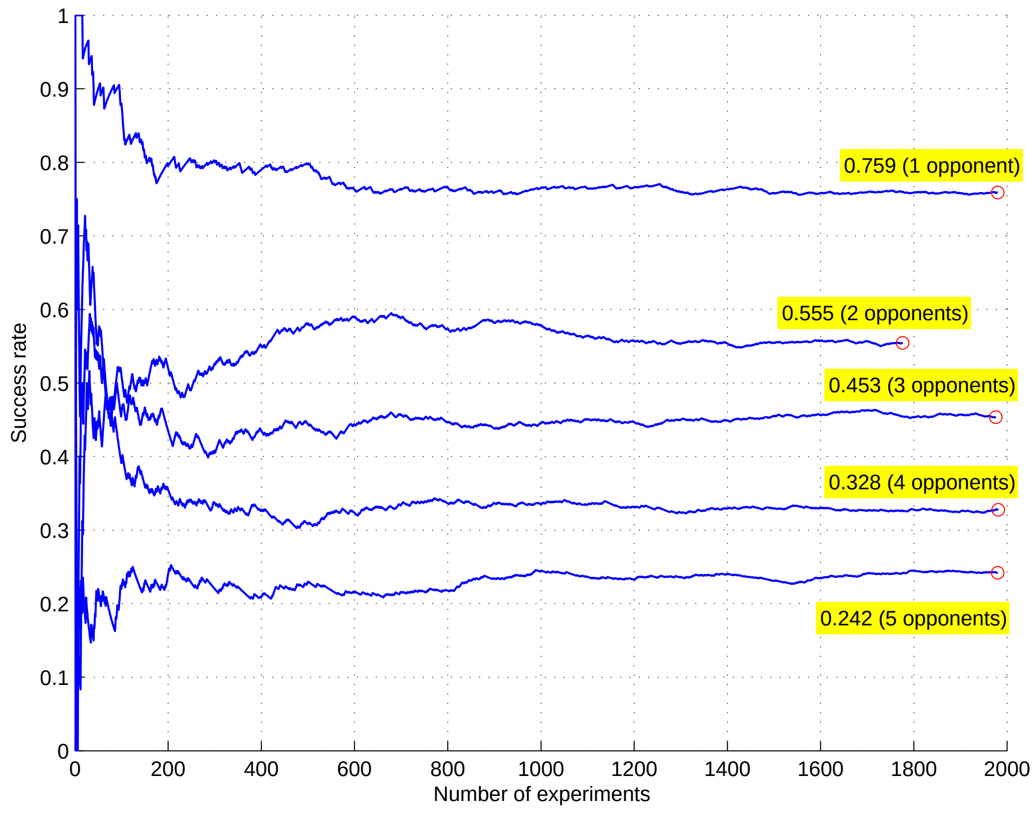


Figure 5.2: Learning experiments using the Givengo plan. The last success rate value and the number of opponent players used is indicated on each graph

followed by Retrieval and Search. The best confidence level we can get for the difference between RL and Retrieval is only 90%. Since the Givengo plan does not require long dribbles and hence take the chance of losing the ball to an intercepting player, search has the chance to score a good number of successes. In contrast, the Centerpass plan requires a long dribble followed by a pass, and that is the main reason why it failed in the Search mode.

Table 5.13 is the comparison of the three application techniques we used in each

	1	2	3	4	5
mean	0.758	0.554	0.456	0.325	0.243
std. dev.	0.001	0.002	0.001	0.001	0.001

Table 5.11: The mean and standard deviation of the last 100 values from the Givengo plan RL experiment

comparison	p-value	t-value	confidence
RL vs Retrieval	0.050	2.772	90.0%
RL vs Search	0.001	9.421	99.9%
Retrieval vs Search	0.034	3.164	95.0%

Table 5.12: Paired t-test results for comparing Givengo plan experiments run in RL, Retrieval, and Search modes

individual opponent scenario. In the 1, 2, and 4 opponent scenarios, RL was no better or worse than Retrieval, and both Retrieval and RL did significantly better than Search. In the 3 and 5 opponent scenarios, on the other hand, RL performed significantly better than RL and Search, but Retrieval did not perform significantly better than Search.

#opponents	#trials	comparison	p-value	t-value	confidence
1	829	RL vs Retrieval	0.114	1.581	–
		RL vs Search	0.000	4.685	95.0%
		Retrieval vs Search	0.001	3.387	95.0%
2	934	RL vs Retrieval	0.052	1.944	–
		RL vs Search	0.000	4.647	95.0%
		Retrieval vs Search	0.004	2.899	95.0%
3	897	RL vs Retrieval	0.000	3.664	95.0%
		RL vs Search	0.000	4.022	95.0%
		Retrieval vs Search	0.373	0.892	–
4	874	RL vs Retrieval	0.153	1.430	–
		RL vs Search	0.000	4.138	95.0%
		Retrieval vs Search	0.007	2.718	95.0%
5	943	RL vs Retrieval	0.012	2.524	95.0%
		RL vs Search	0.001	3.284	95.0%
		Retrieval vs Search	0.462	0.736	–

Table 5.13: Paired t-test results for comparing the performance of RL, Retrieval, and Search modes in each opponent experiment in the Givengo plan

5.3.3 Singlepass Plan

Figure 5.3 shows the learning performance in all five opponent scenarios of the Singlepass plan. Since this plan is simple, the overall performance was relatively higher than in the previous cases we reviewed so far.

Table 5.14 gives the success rates of RL, Retrieval, and Search for all five opponent

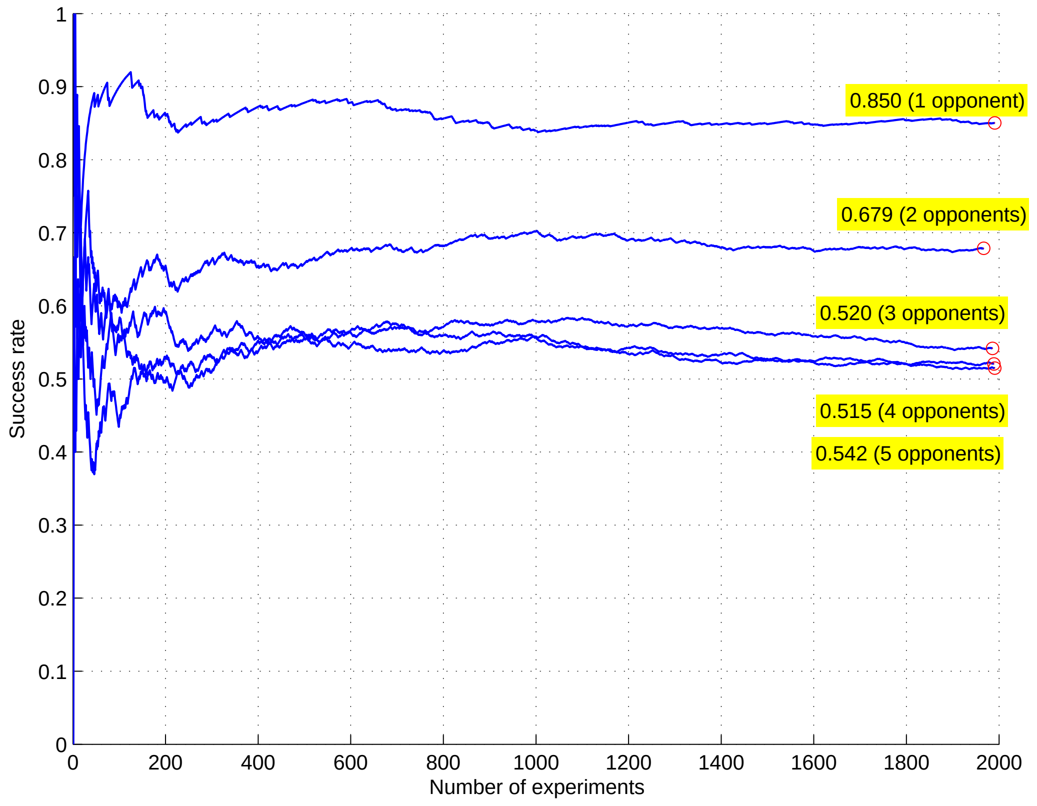


Figure 5.3: Learning experiments using the Singlepass plan. The last success rate value and the number of opponent players used is indicated on each graph

scenarios in the Singlepass plan. From this table, we see that the Search results are consistently better than both RL and Retrieval results.

	1	2	3	4	5
RL	0.851	0.676	0.521	0.515	0.542
Retrieval	0.836	0.674	0.509	0.539	0.551
Search	0.901	0.758	0.603	0.600	0.638

Table 5.14: Success rates of Singlepass plan experiments with $[1 \mathinner{..} 5]$ opponents in RL, Retrieval and Search modes

Table 5.15 shows the mean and the standard deviation of the success rate of the last 100 learning trials for the Singlepass plan. As in the previous two plans, the learning performance converges in all five cases.

According to the paired t-test results given in *Table 5.16*, the RL and Retrieval performances were statistically indistinguishable, but, unlike in the Givengo case, Search

	1	2	3	4	5
mean	0.851	0.676	0.521	0.515	0.542
std. dev.	0.002	0.001	0.001	0.001	0.001

Table 5.15: The mean and standard deviation of the last 100 values from the Singlepass plan RL experiment

performed significantly better than both RL and Retrieval. This is an expected result, since the Singlepass plan only involves a pass, and, therefore, the requirements of this plan do not challenge the search to produce sophisticated solutions. Due to this relative simplicity of the Singlepass plan, RL did not improve the performance of the system over Retrieval.

comparison	p-value	t-value	confidence
RL vs Retrieval	0.944	-0.075	–
RL vs Search	0.001	-10.216	99.9%
Retrieval vs Search	0.000	-12.376	99.9%

Table 5.16: Paired t-test results for comparing Singlepass plan experiments run in RL, Retrieval, and Search modes

Table 5.17 gives the paired t-test results of each of the five opponent scenarios we used in our experiments. When 1, 2, 4, or 5 opponent were used in testing, RL performed equally well as Retrieval, but both RL and Retrieval performed significantly better than Search. In the 3 opponent case, however, RL performed significantly better than Retrieval. In addition, both RL and Retrieval performed significantly better than Search.

5.3.4 UEFA Fourwaypass Plan

Figure 5.4 shows the learning performance of our MuRAL system in the context of the UEFA Fourwaypass plan. Compared to the performance of the system in the case of the previously mentioned three plans, the performance using the UEFA Fourwaypass plan is lower due to its complexity with four steps and three roles.

Table 5.18 lists the success rate of the three techniques we used in evaluating our methodology. Since the Search performance is virtually non-existent, both RL and Retrieval performed much better than Search. Since the UEFA Fourwaypass plan is the most complicated plan we used in this dissertation, agents with only search capability and

#opponents	#trials	comparison	p-value	t-value	confidence
1	995	RL vs Retrieval	0.688	0.401	–
		RL vs Search	0.000	-5.104	95.0%
		Retrieval vs Search	0.000	-5.333	95.0%
2	988	RL vs Retrieval	0.084	1.732	–
		RL vs Search	0.001	-3.492	95.0%
		Retrieval vs Search	0.000	-5.311	95.0%
3	993	RL vs Retrieval	0.004	2.907	95.0%
		RL vs Search	0.010	-2.565	95.0%
		Retrieval vs Search	0.000	-5.564	95.0%
4	992	RL vs Retrieval	0.269	1.106	–
		RL vs Search	0.011	-2.535	95.0%
		Retrieval vs Search	0.000	-3.594	95.0%
5	981	RL vs Retrieval	0.147	1.451	–
		RL vs Search	0.000	-3.753	95.0%
		Retrieval vs Search	0.000	-5.256	95.0%

Table 5.17: Paired t-test results for comparing the performance of RL, Retrieval, and Search modes in each opponent experiment in the Singlepass plan

no training knowledge could not successfully execute this plan.

	1	2	3	4	5
RL	0.403	0.212	0.174	0.119	0.086
Retrieval	0.364	0.194	0.160	0.114	0.063
Search	0.000	0.001	0.000	0.000	0.000

Table 5.18: Success rates of UEFA Fourwaypass plan experiments with $[1 \mathrel{..} 5]$ opponents in RL, Retrieval and Search modes

As *Table 5.19* gives the mean and standard deviation of the success rate of the last 100 learning trials. As this table demonstrates, the learning performance stabilized in all five opponent test scenarios.

Table 5.20 lists the paired t-test results for comparing the overall performance of RL, Retrieval, and Search techniques with the UEFA Fourwaypass plan. We find that RL was significantly better than both Retrieval and Search and Retrieval was significantly better than Search. In the overall comparisons of performance, this is the first plan where we obtain statistically significant results that assert that RL is the best technique to use, followed by Retrieval and finally Search.

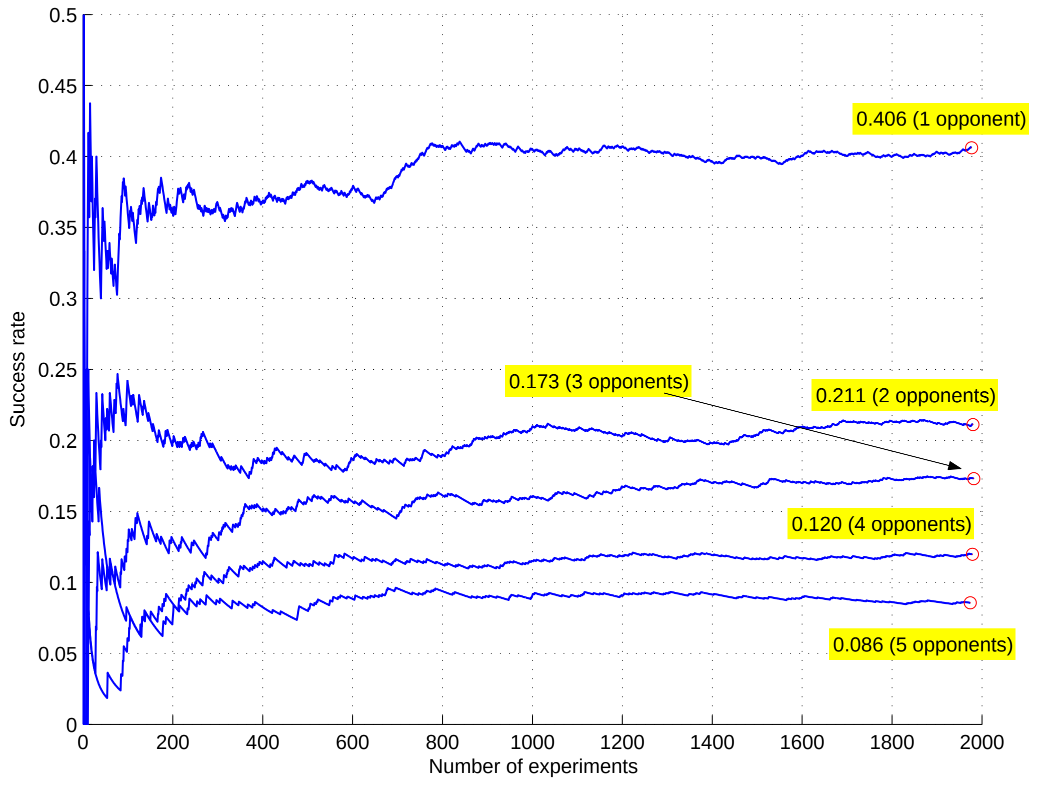


Figure 5.4: Learning experiments using the UEFA Fourwaypass plan. The last success rate value and the number of opponent players used is indicated on each graph

	1	2	3	4	5
mean	0.403	0.212	0.174	0.119	0.086
std. dev.	0.001	0.001	0.001	0.001	0.001

Table 5.19: The mean and standard deviation of the last 100 values from the UEFA Fourwaypass plan RL experiment

Table 5.21 lists the results of the statistical comparisons of RL, Retrieval, and Search in each of the five opponent scenarios. All of the five sets of results are identical. The performance of RL was statistically indistinguishable from that of Retrieval, but both RL and Retrieval performed significantly better than Search. Even though *Table 5.20* shows statistical significance in the overall comparison of RL with Retrieval, when we limit the analysis to individual opponent cases, we find that the RL performs no better or worse than Retrieval.

comparison	p-value	t-value	confidence
RL vs Retrieval	0.027	3.416	95.0%
RL vs Search	0.023	3.577	95.0%
Retrieval vs Search	0.025	3.494	95.0%

Table 5.20: Paired t-test results for comparing UEFA Fourwaypass plan experiments run in RL, Retrieval, and Search modes

#opponents	#trials	comparison	p-value	t-value	confidence
1	881	RL vs Retrieval	0.063	1.864	–
		RL vs Search	0.000	24.495	95.0%
		Retrieval vs Search	0.000	22.467	95.0%
2	868	RL vs Retrieval	0.564	0.577	–
		RL vs Search	0.000	14.481	95.0%
		Retrieval vs Search	0.000	14.001	95.0%
3	869	RL vs Retrieval	0.944	0.070	–
		RL vs Search	0.000	12.912	95.0%
		Retrieval vs Search	0.000	12.857	95.0%
4	835	RL vs Retrieval	0.935	-0.082	–
		RL vs Search	0.000	10.225	95.0%
		Retrieval vs Search	0.000	10.287	95.0%
5	762	RL vs Retrieval	0.081	1.747	–
		RL vs Search	0.000	8.065	95.0%
		Retrieval vs Search	0.000	6.747	95.0%

Table 5.21: Paired t-test results for comparing the performance of RL, Retrieval, and Search modes in each opponent experiment in the UEFA Fourwaypass plan

5.4 Testing of the Effectiveness of Training Knowledge

In this section, our goal is to demonstrate that our agents were sufficiently trained. With few exceptions (See *Table 5.1*), the learning and retrieval experiments we reported in previous sections used knowledge acquired during training using 1000 individual test trials. To show the effect of training, we ran additional training experiments using 500 and 1500 trials for the three plans that learned more than one case in each plan step. These plans were Givengo, Singlepass, and UEFA Fourwaypass.

Tables 5.22, *5.23*, and *5.24* show how many cases were retained for each role for each step of these three plans. As we can observe from these three tables, the more the number of training trials, the more the number of cases for the critical roles in each step. Roles that need to perform ball interception only have a single case since the ball interception RAM does not take on any numerical parameters. In addition, non-critical roles do not learn any cases.

Step#	Role	#Cases (500)	#Cases (1000)	#Cases (1500)
1	A	45	90	141
	B	1	1	1
2	A	465	770	1205
	B	0	0	0
3	A	1	1	1
	B	6	9	10

Table 5.22: Number of cases acquired for each role has in each step of the Givengo plan via training with varying number of test scenarios

Step#	Role	#Cases (500)	#Cases (1000)	#Cases (1500)
1	A	119	210	300
	B	1	1	1

Table 5.23: Number of cases acquired for each role in each step of the Singlepass plan via training with varying number of test scenarios

To demonstrate that we trained our agents sufficiently, we performed retrieval tests for each plan for the 500 and 1500 trial cases, in addition to the retrieval test we already reported on using 1000 training trials. *Table 5.25* summarizes the success rate of these retrieval tests for the three plans using 500, 1000, and 1500 training trials. The results for the 1000 training trials are as reported in earlier sections of this chapter. We include them here for comparison with the 500 and 1500 trial cases. In the case of all three plans, the retrieval performance is comparable in the 500 and 1500 trial cases. However, when we increase the training trials to 1500, the retrieval performance drops. This drop is more apparent in complex plans such as Givengo and UEFA Fourwaypass plan than in the case of the Singlepass plan, which is the simplest plan we used in our testing. We believe this is due to the agents starting to overfit the data when the number of training trails is increased beyond 1000. Therefore, we believe we trained with an appropriate number of cases.

Step#	Role	#Cases (500)	#Cases (1000)	#Cases (1500)
1	A	1	1	1
	B	23	41	79
	C	0	0	0
2	A	58	103	162
	B	0	0	0
	C	1	1	1
3	A	236	581	912
	B	0	0	0
	C	0	0	0
4	A	1	1	1
	B	0	0	0
	C	1	2	2

Table 5.24: Number of cases acquired for each role in each step of the UEFA Fourwaypass plan via training with varying number of test scenarios

Plan	#trials	1	2	3	4	5
Givengo	500	0.797	0.594	0.476	0.362	0.173
Givengo	1000	0.741	0.544	0.382	0.306	0.204
Givengo	1500	0.576	0.370	0.210	0.135	0.052
Singlepass	500	0.810	0.662	0.555	0.545	0.639
Singlepass	1000	0.836	0.674	0.509	0.539	0.551
Singlepass	1500	0.818	0.634	0.518	0.501	0.505
UEFA Fourwaypass	500	0.381	0.202	0.178	0.113	0.051
UEFA Fourwaypass	1000	0.364	0.194	0.160	0.114	0.063
UEFA Fourwaypass	1500	0.126	0.034	0.017	0.009	0.008

Table 5.25: Summary of retrieval tests for three of the plans that learned more than one case per plan step. Each of the three plans was tested using the knowledge obtained from 500, 1000, and 1500 training trials.

5.5 Summary

Table 5.26 summarizes all the overall statistical comparisons for the four plans we designed to demonstrate our MuRAL approach in this dissertation. *Table 5.26* lists the four plans in the order of complexity, where the Singlepass plan is the simplest plan with 2 roles and 1 step and the UEFA Fourwaypass plan is the most complex with 3 roles and 4 steps.

Singlepass (2 roles, 1 step)

	p-value	t-value	confidence
RL vs Retrieval	0.944	-0.075	–
RL vs Search	0.001	-10.216	99.9%
Retrieval vs Search	0.000	-12.376	99.9%

Centerpass (2 roles, 2 steps)

	p-value	t-value	confidence
RL vs Retrieval	0.120	1.974	85.0%
RL vs Search	0.006	5.212	99.0%
Retrieval vs Search	0.006	5.371	99.0%

Givengo (2 roles, 3 steps)

	p-value	t-value	confidence
RL vs Retrieval	0.050	2.772	90.0%
RL vs Search	0.001	9.421	99.9%
Retrieval vs Search	0.034	3.164	95.0%

UEFA Fourwaypass (3 roles, 4 steps)

	p-value	t-value	confidence
RL vs Retrieval	0.027	3.416	95.0%
RL vs Search	0.023	3.577	95.0%
Retrieval vs Search	0.025	3.494	95.0%

Table 5.26: Summary of paired t-test results for all four test plans listed in the top-down order of increasing complexity

The experimental results we presented in this chapter as well as the ones summarized together in *Table 5.26* confirm our theoretical expectations. First, as the complexity of plans increases, the impact of learning grows. Even though the performance of naïve reinforcement learning was statistically indistinguishable from that of Retrieval in Singlepass, Centerpass, and Givengo plans, we see that the confidence of the difference is consistently increasing with increasing plan complexity. In addition, retrieval always performed better than search.

When we compared the performance of RL to Retrieval and Search within the context of each distinct opponent scenario, we found that both RL and Retrieval do statistically better than Search, and RL and Retrieval perform equally well. When Retrieval does not perform statistically significantly better than Search, it turns out that RL does, and, in those instances, RL also performs statistically better than Retrieval (e.g., Givengo plan tests with 3 and 5 opponents, *Table 5.9*). Therefore, statistically, RL does at least as well as Retrieval and sometimes even better.

These results confirm that learning via training is better than behaving reactively, in our case, using search. In other words, we can say that knowing is better than searching. This is a result of each agent learning alternative ways of implementing each subgoal in a plan and therefore allowing the agent to select one of the best solutions that match the current dynamic situation. Since the Singlepass plan was simple, due to being only one step long, search performed the best among the three techniques. However, in plans with more steps and/or more agents, the search performance was always the lowest. Plans with more steps require more dynamic decision-making from the agents, and dynamic reactive behavior requires that each agent have good coverage in their casebases. The only linear result we obtained across all the experiments using our four plans was that increasing the number of opponent players decreased performance. In summary, these results empirically prove our thesis.

◇

Chapter 6

Conclusions

In this dissertation, we studied how to enable a team of non-communicating agents to learn to apply collaborative strategies expressed as high-level plans in complex, dynamic, and unpredictable environments. Real-world dynamic settings pose steep challenges for multiagent problem solving due to the combinatorial explosion of state and action spaces and the difficulty of predicting the environment due to its complexity and dynamics. Even though dynamically changing environments require that agents act reactively, the complexity and unpredictability of such environments require reasoning so that agents can keep their commitments to their shared goals. Therefore, the overall challenge for designing multiagent systems for complex, dynamic, and unpredictable environments rests on the question of how to find a balance between reaction and reasoning such that agents in a multiagent can accomplish their goals despite the constraints in the environment. This question, in turn, boils down to how we deal with intractable search spaces.

The multiagent systems literature has addressed this issue of goal-directed reasoning and reaction largely as a bottom-up learning problem. Bottom-up learning methodologies such as reinforcement learning (RL) create emergent behavior, but they are not good at learning alternative strategies. Due to this limitation, the work reported in the literature focuses on solving multiagent problems that require one-shot decision making. In contrast, our goal in this dissertation was to study alternative methods that would enable teams of agents to pursue multi-step and multi-agent strategies whose execution would naturally span longer times.

Our methodology is based on the idea of *learning by doing* [Anzai and Simon, 1979] and combines the bottom-up and top-down problem-solving approaches without policy learning. We use high-level plans to constrain and focus search towards specific goals and use learning to operationalize each plan and select context-specific implementation to apply to each given situation to accomplish the goals of that plan. Plans initially contain a sequence of steps that describe *what* to do to execute a high-level strategy in a given domain from the individual perspectives of all collaborating agents, but they do not specify *how* a plan is to be carried out in terms of specific executable actions. This separation between description and implementation is essential to our approach, since it facilitates the adaptation of our system to dynamically changing situations. In other words, absolutely necessary conditions and resources become part of the *what* definition in a plan, and what a collaborative team has to adapt to to be able to execute that plan successfully becomes part of the *how* definition.

We refer to the description of individual agent perspectives as *roles*. A role specifies what each agent contributing to the execution of a plan needs to do using preconditions for starting application and postconditions for termination. Therefore, roles are descriptions of the responsibilities that must be taken on by individual agents that wish to collaborate in executing a certain plan. Since our agents do not communicate, agents do not share their role assignments with other agents. Due to this limitation, successful plan application depends on the role assignment of all collaborating agents to match.

To operationalize plans, we train our agents in controlled but situated scenarios where the conditions external to the training agents are changed but kept static throughout each trial. Agents train on one plan at a time, and, for each successful training episode, each agent stores a description of the environment and the sequence of actions it performed using case-based learning. To decide on what sequence of actions to execute in each problem, agents use search. This method of training allows agents to discover solutions with the least intervention from the environment. However, there is no guarantee that the solutions retained during training will work in the full-fledged environment. We use situated application of the knowledge acquired during training as a means to collect feedback from the environment as to which implementations are best suited for each specific situation.

In the next stage of learning, we again situate our agents in a controlled environment, but this time the external factors are no longer kept static. Agents use only the knowledge they retained during training. They use case-based reasoning to match situations to implementations, and they use a naïve form of reinforcement learning to distinguish among the alternatives they learned during training for their current role in a given plan step. This form of reinforcement learning involves strengthening the future selection probability of successful implementations and penalizing that of unsuccessful ones. Then, over many plan application trials, all agents taking part in a given plan autonomously learn to apply plans with increasing performance. Case-based retrieval during application helps deal with the noise associated with matching situations to training knowledge, and naïve RL helps deal with uncertainty in the environment.

In this dissertation, we experimentally showed that, with statistical significance, naïve RL performs at least as well as retrieval. As the complexity of the plans increases, learning has an increasing impact on the system performance such that, beyond a certain plan complexity level, naïve RL performs statistically better than retrieval. In addition, we showed that, except in the case of very simple plans, naïve RL and retrieval are always better than search. Therefore, we conclude that the more roles and steps a plan has, the more learning becomes indispensable in helping improve system performance.

The critical limitation of our problem-solving methodology is that we assume that plans can be written to express complex goals that involve multiple agents. Since our system does not learn plans, it can perform only as well as the extent of its knowledge. It is also possible that there may exist a plan to solve a problem, but the role a given agent assumed in that plan may not have the application knowledge needed to execute that role. In addition, there may exist no plans to solve particular problems in a domain. Hence, we may need a large set of plans to enable a multiagent system to operate continuously in a given dynamic, complex, and uncertain environment.

In our work, we have not addressed any issues that may arise when a multiagent system like ours is to be migrated from the simulation environment to a real-world setting. That there exist challenges when this migration is done is well-known [Gat, 1997]. Even though the simulator we used modeled sensors and actuators to have noise, it is conceivable that, in an actual application, sensors may totally fail or actuators may not work consistently. In other words, the nature of failures may be more severe than in simulation, and recovery from failure may be more difficult. Since it is virtually impossible to prove the completeness of a multiagent system designed for complex, dynamic, and uncertain domains, plans themselves may fail due to lack of critical knowledge, as we suggested.

The many challenges of working with multiagent systems are still apparent in the simulated environment. For example, in our experience, we found that the perception problem is one of the most critical challenges. One of the critical decision points a player repeatedly visits in soccer has to with deciding whether another player has possession of the ball. At first try, one may think it sufficient to know that a player has the ball as long as the ball is near that player and is practically stopped. However, the complexity of soccer does not lend itself to such simplification. For instance, if a player is turning the ball around itself in order to give a pass, the ball during that action will have non-zero speed. Therefore, relying partially on the ball having near-zero speed to decide on ball possession will lead to detection failures. Since many decisions at plan level depend on such lower-level decisions, multiagent systems tend to be brittle. Moreover, real-world occurrences are almost never instantaneous. Instead, they span over time, and this means that their detection must also span over time.

6.1 Future Work

In the light of our work, we discuss three issues of interest for future work:

1. The ultimate goal of our line of work is to automate the creation of fully-autonomous teams of agents that can execute plans in succession to accomplish higher-level goals

that may also be specified in plans themselves. Besides the fundamental difficulties of sensing and acting, this operation requires coherence among multiple agents, and, at the minimum, it entails the following:

(a) Each agent in a collaborative group must select the same plan to execute at the same time as other agents that it believes it can collaborate with.

(b) The role assignments/mappings performed from each agent's perspective must be consistent with those of other agents in the same collaborative group.

(c) Each step of the selected plan must be performed in coordination with all other agents contributing to it.

How we ensure that role assignments are consistent is a difficult question to answer without involving communication among agents. As we have demonstrated in this dissertation, agents operating in physical environments can, in theory, only use their sensory capabilities to help match plans to situations. Even though we had a single plan to use in training and application phases, that plan still had to be matched to the situation. In addition, the controlled nature of our experiments also implicitly defined when the execution of a plan could start. However, it may be more difficult to match plans to situations in a regular setting where there are no restrictions imposed as in the learning approach we used. In addition, even though an agent can only assign one agent per role, with more agents in the environment than needed for executing any given plan, there may be multiple agents that can be candidates for a given role. Without communication, it is difficult to guarantee that the internal role assignment of one agent will match that of other agents that are in position to contribute to the execution of the plan. Therefore, conflicts might arise. Resolving these conflicts may require negotiation. However, negotiating during time-critical activity brings its own disadvantages to solving multiagent problems in highly dynamic environments where communication costs are high.

2. In a multiagent system, it is difficult to define the completeness of the knowledge each agent uses since there are infinitely many possible situations it is expected to

handle. Therefore, situations in which an agent fails to find applicable implementation in its knowledge base are opportunities for further training and learning. An agent can record a description of the environment in a case as it would during training, except that the case would contain no application knowledge. Such cases can be marked as potential sources for more learning and agents involved in such scenarios can be trained over those cases.

3. In MuRAL, the training phase uses search to acquire application knowledge that is expressed in terms of high-level reactive actions. Some of these actions take on continuous-valued arguments. For example, the ball dribbling action, `dribble-ball` (see *Section 4.2.2*) requires (x,y) coordinate values as argument. In our current approach, we do not generalize such continuous argument values while merging plans (see *Section 3.6.1*). We only check for identity of corresponding argument values up to the second decimal point. However, there may be many solutions stored in plan casebases that have the identical sequence of actions in terms of the type of the actions involved but with slightly different action arguments. By generalizing such type-identical action sequences, we can reduce all action sequences with statistically similar continuous arguments into a single action sequence. One possible approach for generalizing action sequences is to divide the region around a given player into grids and consider all values that fall within a grid as similar.

6.2 Summary

The experimental results we obtained validate our thesis that a multiagent system with non-communicating agents operating in highly dynamic, complex, and uncertain environments can learn to improve its plan execution performance. To achieve this goal, we introduced a knowledge-based approach to solving multiagent problems using symbolic plans. We showed that separating conditions that a team of agents can have control over from those that they cannot exercise any reasonable control over greatly facilitates learning and therefore adaptation of behavior to dynamic changes.

We used plans to constrain search from top-down and used learning to build specific implementations in a bottom-up fashion, hence taking advantage of both approaches for problem solving. We introduced a learning technique that combines case-based learning with a naïve form of reinforcement learning that proved to be effective in dynamic settings. We contributed a training and learning process that enables autonomous learning in multiagent systems with non-communicating agents. We contributed a fully-implemented system that exhibits true multiagent behavior that is collaborative. In this regard, our plan representation defines the parameters of the collaboration required. In contrast to plans in traditional planning systems, our plans facilitate dynamic adaptive autonomous behavior and autonomous learning from the local perspectives of individual agents.

◇

Bibliography

Agnar Aamodt and Enric Plaza. Case-based reasoning : Foundational issues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994. ◇ pp: 81, 82

M. Adelantado and S. De Givry. Reactive/anytime agents: Towards intelligent agents with real-time performance. In *Working Notes of the IJCAI Workshop on Anytime Algorithms and Deliberation Scheduling*, Montreal, Canada, 1995. ◇ pp: 21

Philip E. Agre and David Chapman. Pengi: An implementation of a theory of activity. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI 1987)*, pages 268–272, 1987. ◇ pp: 13, 22

Philip E. Agre and David Chapman. What are plans for? Technical Report AI-MEMO 1050a, MIT AI Lab, October 1989. Revised version (original September 1988). ◇ pp: 22

David W. Aha. Case-based learning algorithms. In *Proceedings of the DARPA Case-Based Reasoning Workshop*, pages 147–158, Washington, D.C., USA, 1991. Morgan Kaufmann. ◇ pp: 81, 83

James Allen, James Hendler, and Austin Tate, editors. *Readings in Planning*. The Morgan Kaufmann Series in Representation and Reasoning. Morgan Kaufmann, 1990. ◇ pp:

José A. Ambros-lngerson and Sam Steel. Integrating planning, execution and monitoring. In *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 83–88, Saint Paul, Minnesota, USA, August 1988. AAAI Press/MIT Press. (Reprinted in [Allen et al., 1990], pages 735–740). ◇ pp: 14, 78

Yuichiro Anzai and Herbert A. Simon. The theory of learning by doing. *Psychological Review*, 86(2):124–140, 1979. ◇ pp: 6, 168

Ronald C Arkin and Tucker Balch. AuRA: Principles and practice in review. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2/3):175–189, April 1997. ◇ pp: 26, 51

Shawn Arseneau, Wei Sun, Changpeng Zhao, and Jeremy R. Cooperstock. Inter-layer learning towards emergent cooperative behavior. In *The Seventeenth National Conference on Artificial Intelligence, Twelfth Innovative Applications of AI Conference (AAAI/IAAI 2000)*, pages 3–8, 2000. ◇ pp: 129

Tucker Balch. Clay: Integrating motor schemas and reinforcement learning. Technical Report GIT-CC-97-11, Georgia Institute of Technology. College of Computing, 1997a. ◇ pp: 55

Tucker Balch. Learning roles: Behavioral diversity in robot teams. Technical Report GIT-CC-97-12, Georgia Institute of Technology. College of Computing, 1997b. ◇ pp: 6, 55, 99

Scott Benson. Inductive learning of reactive action models. In Armand Prieditis and Stuart J. Russell, editors, *Proceeding of the Twelfth International Conference on Machine Learning (ICML 1995)*, pages 47–54. Morgan Kaufmann, 1995. ◇ pp: 47

Tommaso F. Bersano-Begey, Patrick G. Kenny, and Edmund H. Durfee. Multi-agent teamwork, adaptive learning and adversarial planning in Robocup using a PRS architecture. Unpublished manuscript. Personal communication, 1998. ◇ pp: 6

Alan H. Bond and Les Gasser, editors. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann, 1988. ◇ pp: 14

Blai Bonet and Héctor Geffner. Heuristic Search Planer 2.0. *AI Magazine*, 22(3):77–80, Fall 2001a. ◇ pp: 18

Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129 (1–2):5–33, June 2001b. ◇ pp: 99

Blai Bonet, Gábor Loerincs, and Hector Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1997)*, pages 714–719. AAAI Press/The MIT Press, July 1997. ◇ pp: 18, 42

Michael Bowling, Brett Browning, and Manuela Veloso. Plays as effective multiagent plans enabling opponent-adaptive play selection. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, page (To appear), 2004. ◇ pp: 129, 130

Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(4):349–355, 1988. ◇ pp: 27, 31, 44, 68, 78

Frances Brazier, Barbara Dunin-Keplicz, Nick R. Jennings, and Jan Treur. Formal specification of multi-agent systems: a real–world case. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 25–32, San Francisco, CA, USA, June 1995. ◇ pp: 50

Will Briggs and Diane Cook. Anytime planning for optimal tradeoff between deliberative and reactive planning. In *Proceedings of the 1999 Florida AI Research Symposium (FLAIRS 1999)*, pages 367–370, 1999. ◇ pp: 40, 99

Rodney Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(Issue 1):14–23, March 1986. ◇ pp: 13, 23, 30, 34

Rodney A. Brooks. A robust layered control system for a mobile robot. Technical Report AIM-864, Massachusetts Institute of Technology, September 1985. ◇ pp: 24

Rodney A. Brooks. A robot that walks: Emergent behaviors from a carefully evolved network. Ai memo 1091, MIT, 1989. ◇ pp: 13, 24

Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47(1–3):139–159, January 1991. ◇ pp: 13, 24, 25, 31

Hans-Dieter Burkhard, Markus Hannebauer, and Jan Wendler. AT Humboldt – development, practice and theory. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Artificial Intelligence*, pages 357–372. Springer, 1998. ◇ pp: 54

Hans-Dieter Burkhard, Jan Wendler, Thomas Meinert, Helmut Myritz, and Gerd Sander. AT Humboldt in RoboCup-99. In Manuela M. Veloso, Enrico Pagello, and Hiroaki Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, volume 1856 of *Lecture Notes in Artificial Intelligence*, pages 542–545. Springer, 2000. ◇ pp: 54

Brahim Chaib-draa, Bernard Moulin, and P. R. et Millot Mandiau. Trends in distributed artificial intelligence. *Artificial Intelligence Review*, 6(1):35–66, 1992. ◇ pp: 14

David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987. ◇ pp: 16

Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Pat Riley, Timo Steffens, Yi Wang, and Xiang Yin. RoboCup Soccer Server. Manual (for SoccerServer version 7.07 and later), , August 2002. ◇ pp: 4, 58, 59, 63, 67, 118, 131, 144, 226

CMUnited source code. (Last access on 04/29/2004). URL `http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/robosoccer/www/simulator`. ◇ pp: 131

Paul R. Cohen. *Empirical Methods for Artificial Intelligence*. The MIT Press, 1995. ◇ pp: 151

Silvia Coradeschi and Lars Karlsson. A role-based decision-mechanism for teams of reactive and coordinating agents. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Artificial Intelligence*, pages 112–122. Springer, 1998. ◇ pp: 129

Mark d'Inverno, David Kinny, Michael Luck, and Michael Wooldridge. A formal specification of dMARS. In *Intelligent Agents IV: Proceedings of Fourth International*

Workshop on Agent Theories, Architectures, and Languages (ATAL '97), pages 155–176, 1997. ◇ pp: 27

J. E. Doran, S. Franklin, N. R. Jennings, and T. J. Norman. On cooperation in multi-agent systems. *Knowledge Engineering Review*, 12(3):309–314, 1997. ◇ pp: 38

Edmund H. Durfee, Victor R. Lesser, and Daniel D. Corkill. Trends in cooperative distributed problem solving. *IEEE Transactions on Knowledge and Data Engineering*, 1(1): 63–83, 1989. ◇ pp: 127

Edmund H. Durfee and Jeffrey S. Rosenschein. Distributed problem solving and multi-agent systems: Comparisons and examples. In *Papers from the Thirteenth International Workshop on Distributed Artificial Intelligence (Technical Report WS-94-02)*, pages 94–104. AAAI Press, 1994. ◇ pp: 14

Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994)*, volume 2, pages 1123–1128, Seattle, Washington, USA, 1994. AAAI Press. ◇ pp: 16, 17

Innes A. Ferguson. TouringMachines: Autonomous agents with attitudes. *IEEE Computer*, 25(5):51–55, May 1992a. ◇ pp: 34

Innes Andrew Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, UComputer Laboratory, University of Cambridge, UK, October 1992b. (Also Technical Report 273, Computer Laboratory, University of Cambridge, UK). ◇ pp: 34

Gabriel J. Ferrer. *Anytime Replanning Using Local Subplan Replacement*. PhD thesis, University of Virginia, May 2002. ◇ pp: 40

Richard Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4):189–208, 1971. (Reprinted in [Allen et al., 1990], pages 88–97). ◇ pp: 12, 16

James R. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Department of Computer Science, Yale University, January 1989. ◇ pp: 21

R. James Firby. An investigation into reactive planning in complex domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI 1987)*, pages 202–206. Morgan Kaufmann, July 1987. ◇ pp: 13, 21

R. James Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems (AIPS 1994)*, pages 49–54, 1994. ◇ pp: 21

Douglas H. Fisher. Knowledge acquisition via incremental conceptual clustering. *Machine Learning*, 2(2):139–172, 1987. ◇ pp: 51

Andrew Garland and Richard Alterman. Multiagent learning through collective memory. In Sandip Sen, editor, *Papers from the AAAI Symposium on Adaptation, Coevolution and Learning in Multiagent Systems (Technical Report SS-96-01)*, pages 33–38, Stanford University, California, USA, March 1996. ◇ pp: 50, 51

Andrew Garland and Richard Alterman. Learning cooperative procedures. In Ralph Bergmann and Alexander Kott, editors, *AIPS Workshop on Integrating Planning, Scheduling, and Execution in Dynamic and Uncertain Environments (Technical Report WS-98-02)*, pages 54–61. AAAI Press, 1998. ◇ pp: 50, 51

Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In William Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence*, pages 809–815, San Jose, CA, July 1992. MIT Press. ISBN 0-262-51063-4. ◇ pp: 30

Erann Gat. On the role of stored internal state in the control of autonomous mobile robots. *AI Magazine*, 14(1):64–73, Spring 1993. ◇ pp: 30

Erann Gat. Towards principled experimental study of autonomous mobile robots. In Oussama Khatib and John Kenneth Salisbury Jr., editors, *Proceedings of the 4th International*

Symposium on Experimental Robotics (ISER 1995), volume 223 of *Lecture Notes in Control and Information Sciences*, pages 390–401, Stanford, California, USA, 1997. Springer. ⋄ pp: 170

Erann Gat. On three-layer architectures. In David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1998. ⋄ pp: 27

Michael P. Georgeff. Planning. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 525–49. Morgan Kaufmann, 1990. ⋄ pp: 16, 17

Michael P. Georgeff and François Félix Ingrand. Monitoring and control of spacecraft systems using procedural reasoning. Technical Note 03, Australian Artificial Intelligence Institute, Melbourne, Australia, 1989. ⋄ pp: 28

Michael P. Georgeff and Amy L. Lansky. Rective reasoning and planning. In *Proceedings of Sixth National Conference on Artificial Intelligence (AAAI 1987)*, pages 677–682, 1987. ⋄ pp: 13, 27

Mike Georgeff, Barney Pell, Martha Pollack, Milind Tambe, and Mike Wooldridge. The belief-desire-intention model of agency. In Jörg Müller, Munindar P. Singh, and Anand S. Rao, editors, *Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)*, volume 1555 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 1999. ⋄ pp: 27

Melinda T. Gervasio. Learning general completable reactive plans. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI 1990)*, pages 1016–1021. AAAI Press/The MIT Press, 1990. ⋄ pp: 48

John J. Grefenstette and Connie Loggia Ramsey. An approach to anytime learning. In *Proceedings of the Ninth International Machine Learning Workshop (ML 1992)*, pages 189–195, 1992. ⋄ pp: 39, 40

Peter Haddawy. Focusing attention in anytime decision-theoretic planning. In *Working Notes of the 1995 IJCAI Workshop on Anytime Algorithms and Deliberation Scheduling*, pages 73–84, August 1995. ◇ pp: 40

Karen Zita Haigh and Manuela M. Veloso. Planning, execution and learning in a robotic agent. In Reid G. Simmons, Manuela M. Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS 1998)*, pages 120–127, 1998. ◇ pp: 48

Kristian J. Hammond. CHEF: A model of case-based planning. In Tom Kehler and Stan Rosenschein, editors, *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI 1986)*, volume 1, pages 267–271, Los Altos, CA, August 1986. Morgan Kaufmann. ◇ pp: 13

Kristian J. Hammond. *Case-Based Planning: viewing planning as a memory task*. Perspectives in AI. Academic Press, Boston, MA, 1989. ◇ pp: 13

Thomas Haynes, Kit Lau, and Sandip Sen. Learning cases to compliment rules for conflict resolution in multiagent systems. In Sandip Sen, editor, *Papers from the AAAI Symposium on Adaptation, Coevolution and Learning in Multiagent Systems (Technical Report SS-96-01)*, pages 51–56, Stanford University, California, USA, 1996. ◇ pp:

Thomas Haynes and Sandip Sen. Adaptation using cases in cooperative groups. In Ibrahim F. Imam, editor, *Papers from the AAAI Workshop on Intelligent Adaptive Agents (Technical Report WS-96-04)*, pages 85–93, 1996a. (Also see [Haynes et al., 1996; Haynes and Sen, 1996b]). ◇ pp: 50

Thomas Haynes and Sandip Sen. Learning cases to resolve conflicts and improve group behavior. In Milind Tambe and Piotr Gmytrasiewicz, editors, *Papers from the AAAI Workshop on Agent Modeling (Technical Report WS-96-02)*, pages 46–52, Portland, OR, 1996b. ◇ pp:

Henry Hexmoor and Donald Nute. Methods for deciding what to do next and learning. AI Research Report AI-1992-01, Artificial Intelligence Center, The University of Georgia, 1992. ⋄ pp: 16, 21

Okhtay Ilghami, Dana S. Nau, Héctor Muñoz-Avila, and David W. Aha. CaMeL: Learning method preconditions for HTN planning. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling (AIPS 2002)*, pages 131–142, 2002. ⋄ pp: 47

François Félix Ingrand and Vianney Coutance. Real-time reasoning using procedural reasoning. Technical Note 93104, LAAS, 1993. ⋄ pp: 28

Toru Ishida. Real-time search for autonomous agents and multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 1(2):139–167, October 1998. ⋄ pp: 41, 42

N. R. Jennings. Coordination techniques for distributed artificial intelligence. In Greg M. P. O'Hare and N. R. Jennings, editors, *Foundations of Distributed Artificial Intelligence*, pages 187–210. John Wiley & Sons, 1996. ⋄ pp: 36, 38

Christoph G. Jung and Klaus Fischer. Methodological comparison of agent models. Research Report RR-98-01, German Research Center for Artificial Intelligence (DFKI GmbH), October 1998. ⋄ pp: 33

Leslie P. Kaelbling. A situated-automata approach to the design of embedded agents. *SIGART Bulletin*, 2(4):85–88, 1991. ⋄ pp: 23

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. ⋄ pp: 45

Subbarao Kambhampati. Mapping and retrieval during plan reuse: A validation structure based approach. In *Proceedings of the Eight National Conference on Artificial Intelligence (AAAI 1990)*, pages 170–175. AAAI Press/The MIT Press, 1990. ⋄ pp: 13

Subbarao Kambhampati and James A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):193–258, 1992. ⋄ pp: 13

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In *IJCAI 1995 Workshop on Entertainment and AI/ALife*, pages 19–24, 1995. ◇ pp: 6

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The robot world cup initiative. In *Proceedings of the First International Conference on Autonomous Agents*, pages 340–347, Marina Del Rey, California, USA, February5–8 1997a. ACM Press. ◇ pp: 4, 6, 56, 67, 118

Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. RoboCup: A challenge problem for AI. *AI Magazine*, 18(1):73–85, Spring 1997b. ◇ pp: 4, 6, 56, 67, 118

Hiroaki Kitano, Milind Tambe, Peter Stone, Manuela Veloso, Silvia Coradeschi, Eiichi Osawa, Hitoshi Matsubara, Itsuki Noda, and Minoru Asada. The RoboCup synthetic agent challenge 97. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 24–29, Nagoya, Japan, 1997c. Morgan Kaufmann. (Vol. 1). ◇ pp: 56

Sven Koenig. Minimax real-time heuristic search. *Artificial Intelligence*, 129(1–2):165–197, 2001. ◇ pp: 42

Richard E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33(1): 65–88, September 1987. ◇ pp: 80

Richard E. Korf. Real-time heuristic search. *Artificial Intelligence*, 42(2-3):189–211, March 1990. ◇ pp: 39, 41

Kostas Kostiadis and Huosheng Hu. Reinforcement learning and co-operation in a simulated multi-agent system. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 1999)*, volume 2, pages 990–995, October 1999. ◇ pp: 87

Kostas Kostiadis and Huosheng Hu. KaBaGe-RL: Kanerva-based generalisation and reinforcement learning for possession football. In *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*, volume 1, pages 292–297, 2001. ◇ pp: 5, 55, 129

Gregory Kuhlmann and Peter Stone. Progress in learning 3 vs. 2 keepaway. In *RoboCup-2003: Robot Soccer World Cup VII*, 2003. To appear. ◇ pp: 55, 129

Nicholas Kushmerick, Steve Hanks, and Daniel S. Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1073–1078, Seattle, Washington, USA, 1994. AAAI Press. ◇ pp: 20

J. Brian Lee, Maxim Likhachev, and Ronald C. Arkin. Selection of behavioral parameters: Integration of discontinuous switching via case-based reasoning with continuous adaptation via learning momentum. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2002)*, volume 2, pages 1275–1281, May 2002. ◇ pp: 51

Maxim Likhachev, M Kaess, and Ronald C. Arkin. Learning behavioral parameterization using spatio-temporal case-based reasoning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2002)*, volume 2, pages 1282–1289, May 2002. ◇ pp: 51

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the Eleventh International Conference on Machine Learning (ICML 1994)*, pages 157–163, 1994. ◇ pp: 52

Sean Luke. Evolving Soccerbots: A retrospective. In *Proceedings of 12th Annual Conference of the Japanese Society for Artificial Intelligence (JSAI 1998)*, 1998. ◇ pp: 6, 55

Sean Luke. *Issues in Scaling Genetic Programming: Breeding Strategies, Tree Generation, and Code Bloat*. PhD thesis, Department of Computer Science, University of Maryland, 2000. ◇ pp:

Sean Luke, Charles Hohn, Jonathan Farris, Gary Jackson, and James A. Hendler. Co-evolving soccer softbot team coordination with genetic programming. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*, pages 398–411. Springer, 1998. (Also see [Luke, 2000, Chapter 5]). ◇ pp: 6, 55

Alan K. Mackworth. On seeing robots. In Anup Basu and Xiaobo Li, editors, *Computer Vision: Systems, Theory and Applications*, pages 1–13. World Scientific, 1993. ◇ pp: 56

Pattie Maes. The dynamics of action selection. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI 1989)*, volume 2, pages 991–997, 1989. ◇ pp: 24

Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence*, 55(2):311–365, June 1992. ◇ pp: 44, 129

Rajbala Makar, Sridhar Mahadevan, and Mohammad Ghavamzadeh. Hierarchical multi-agent reinforcement learning. In *Proceedings of the Fifth International Conference on Autonomous Agents (Agents 2001)*, pages 246–253, 2001. ◇ pp: 52

Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys*, 26(1):87–119, March 1994. ◇ pp: 37

Maja J Matarić. A comparative analysis of reinforcement learning methods. Memo 1322, MIT AI Lab, October 1991. ◇ pp: 4, 71, 127

Maja J Matarić. Behavior-based control: Main properties and implications. In *Proceedings of the IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, pages 46–54, 1992. (See [Matarić, 1997a]). ◇ pp: 23, 24

Maja J. Mataric. *Interaction and Intelligent Behavior*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 1994. (Also MIT Artificial Intelligence Lab, Technical Report AITR-1495). ◇ pp: 128

Maja J. Matarić. Learning in multi-robot systems. In Gerhard Weiß and Sandip Sen, editors, *Adaption and Learning in Multi-Agent Systems*, volume 1042 of *Lecture Notes in Computer Science*, pages 152–163, 1996. ◇ pp: 4, 6, 31, 71, 117, 127, 128

Maja J Matarić. Behavior-based control: Examples from navigation, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence (Special issue on Software Architectures for Physical Agents)*, 9(2–3):323–336, 1997a. ◇ pp:

Maja J Matarić. Reinforcement learning in the multi-robot domain. *Autonomous Robots*, 4 (1):73–83, March 1997b. ◇ pp: 47

Maja J Matarić. Coordination and learning in multirobot systems. *IEEE Intelligent Systems*, 13(Issue 2):6–8, March/April 1998. ◇ pp: 99

Hitoshi Matsubara, Itsuki Noda, and Kazuo Hiraki. Learning of cooperative actions in multi-agent systems: A case study of pass play in soccer. In Sandip Sen, editor, *Papers from the AAAI Symposium on Adaptation, Coevolution and Learning in Multiagent Systems (Technical Report SS-96-01)*, pages 63–67, Stanford University, California, USA, March 1996. ◇ pp: 52

William Mendenhall and Terry Sincich. *Statistics for Engineering and the Sciences*. Dellen Publishing Company, third edition, 1992. ◇ pp: 151

Artur Merke and Martin A. Riedmiller. Karlsruhe Brainstormers - A reinforcement learning approach to robotic soccer. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Artificial Intelligence*, pages 435–440. Springer, 2002. ISBN 3-540-43912-9. ◇ pp: 5, 6

Merriam-Webster. Dictionary. (Last access on 04/29/2004). URL `http://www.m-w.com`. ◇ pp: 36

David J. Musliner, Edmund H. Durfee, and Kang G. Shin. CIRCA: A cooperative intelligent real time control architecture. *IEEE Transactions on Systems, Man, and*

Cybernetics, 23(6):1561–1574, November-December 1993. (Also Technical Report CS-TR-3157, University of Maryland, College Park). ◇ pp: 39, 42

David John Musliner. CIRCA: The cooperatice intelligent real-time control architecture. Technical Report CS-TR-3157, University of Maryland, College Park, October 1993. ◇ pp: 39

Karen L. Myers. Strategic advice for hierarchical planners. In L. C. Aiello, J. Doyle, and S. C. Shapiro, editors, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996)*, pages 112–123. Morgan Kaufmann Publishers, 1996. ◇ pp: 4, 72, 126

Karen L. Myers. Abductive completion of plan sketches. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1997)*, pages 687–693. AAAI Press, 1997. ◇ pp: 4, 72, 126

Allen Newell. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990. ◇ pp: 19, 20

Allen Newell and Herbert A. Simon. GPS, a program that simulates human thought. In Edward A. Feigenbaum and Julian Feldman, editors, *Computers and Thought*, pages 279–296. McGraw-Hill, 1963. ◇ pp: 11

Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, March 1976. ◇ pp: 12, 19

Itsuki Noda. Soccer server : A simulator of RoboCup. In *Proceedings of AI Symposium '95, Japanese Society for Artificial Intelligence*, pages 29–34, December 1995. ◇ pp: 57

Itsuki Noda, Hitoshi Matsubara, Kazuo Hiraki, and Ian Frank. Soccer Server: a tool for research on multiagent systems. *Applied Artificial Intelligence*, 12(2-3):233–250, 1998. (Also Electrotechnical Labaratory Technical Report TR-97-11). ◇ pp: 57

Itsuki Noda and Peter Stone. The RoboCup Soccer Server and CMUnited: Implemented infrastructure for MAS research. In Thomas Wagner and Omer F. Rana, editors, *International Workshop on Infrastructure for Multi-Agent Systems (Agents 2000)*, volume 1887 of *Lecture Notes in Computer Science*, pages 94–101, Barcelona, Spain, 2001. Springer. ⋄ pp: 57, 87

Gary B. Parker and H. Joseph Blumenthal. Punctuated anytime learning for evolving a team. In *Proceedings of the World Automation Congress (WAC 2002)*, volume 14 (Robotics, Manufacturing, Automation and Control), pages 559–566, June 2002. ⋄ pp: 6, 44, 54

Bernard Pavard and Julie Dugdale. The contribution of complexity theory to the study of socio-technical cooperative systems. *New England Complex Systems Institute electronic journal (InterJournal), http://www.interjournal.org/*, 2000. (Manuscript #335). ⋄ pp: 68

D. W. Payton, D. Keirsey, D. M. Kimble, J. Krozel, and J. K. Rosenblatt. Do whatever works: A robust approach to fault-tolerant autonomous control. *Journal of Applied Intelligence*, 3:226–250, 1992. ⋄ pp: 24

Martha Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI 1990)*, pages 183–189, 1990. ⋄ pp: 32

J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, USA, 1993. ⋄ pp: 53

Anand S. Rao and Michael P. Georgeff. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, USA, June 1995. ⋄ pp: 27, 54

Luís Paulo Reis and Nuno Lau. Fc portugal team description: Robocup 2000 simulation league champion. In Peter Stone, Tucker Balch, and Gerhard Kraetzschmar, editors, *RoboCup 2000: Robot Soccer World Cup IV*, volume 2019 of *Lecture Notes in Artificial Intelligence*, pages 29–40. Springer, 2001. ⋄ pp: 87

Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, New York, second edition, 1991. ◇ pp: 80, 83, 84

Martin Riedmiller and Artur Merke. Using machine learning techniques in complex multi-agent domains. In Reiner Kühn, Randolf Menzel, Wolfram Menzel, Ulrich Ratsch, Michael M. Richter, and Ion-Olimpiu Stamatescu, editors, *Adaptivity and Learning: An Interdisciplinary Debate*, pages 311–328. Springer, 2003. ◇ pp: 44, 45, 55, 56

Martin Riedmiller, Artur Merke, Andreas Hoffmann, Manuel Nickschas, Daniel Withopf, and Franziska Zacharias. Brainstormers 2002 - team description. In *RoboCup 2002: Robot Soccer World Cup VI*, Pre-Proceedings, Fukuoka, Japan, 2002. ◇ pp: 102, 103

Paul S. Rosenbloom, John E. Laird, and Allen Newell. *The Soar Papers: Research on Artificial Intelligence*, volume volume 1 and 2. MIT Press, Cambridge, Massachusetts, 1993. ◇ pp: 19, 20

Stanley J. Rosenschein and Leslie Pack Kaelbling. A situated view of representation and control. *Artificial Intelligence*, 73(1–2):149–173, 1995. ◇ pp: 23

Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5(2): 115–135, 1974. (Reprinted in [Allen et al., 1990], pages 98–108). ◇ pp: 12

Earl D. Sacerdoti. The non-linear nature of plans. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI 1975)*, pages 206–214, 1975. (Reprinted in [Allen et al., 1990], pages 171–184). ◇ pp: 16

Michael K. Sahota. Reactive deliberation: An architecture for real-time intelligent control in dynamic environments. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 1303–1308. AAAI Press, 1994. ◇ pp: 56

Hüseyin Sevay and Costas Tsatsoulis. Multiagent reactive plan application learning in dynamic environments. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2002)*, pages 839–840, July 2002. ◇ pp: 72

Herbert A. Simon. *Models of Bounded Rationality*. MIT Press, 1982. ◇ pp: 39

Lin Padgham Simon Ch'ng. From roles to teamwork: A framework and architecture. *Applied Artificial Intelligence*, 12(2/3):211–231, 1998a. ◇ pp: 6

Lin Padgham Simon Ch'ng. Team description: Building teams using roles, responsibilities, and strategies. In Hiroaki Kitano, editor, *RoboCup-97: Robot Soccer World Cup I*, volume 1395 of *Lecture Notes in Computer Science*, pages 458–466. Springer, 1998b. ◇ pp: 6, 130

William D. Smart and Leslie Pack Kaelbling. Practical reinforcement learning in continuous spaces. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 903–910, Stanford, CA, USA, 2000. Morgan Kaufmann. ◇ pp: 5, 46

John A. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 21(10): 10–19, October 1988. ◇ pp: 39

Mark Stefik. Planning with constraints (MOLGEN: Part 1). *Artificial Intelligence*, 16(2): 111–140, 1981. (Reprinted in [Allen et al., 1990], pages 171–184). ◇ pp: 13, 16

Peter Stone. *Layered Learning in Multi-Agent Systems*. PhD thesis, Carnegie Mellon University, December 1998. (Also Technical Report CMU-CS-98-187, Computer Science Department, Carnegie Mellon University). ◇ pp: 7, 52, 53, 55, 72, 102, 127

Peter Stone and Richard S. Sutton. Scaling reinforcement learning toward RoboCup soccer. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 537–544, Stanford, CA, USA, 2001. Morgan Kaufmann. ◇ pp: 4, 5, 44, 45, 55, 128, 129

Peter Stone and Richard S. Sutton. Keepaway soccer: A machine learning testbed. In Andreas Birk, Silvia Coradeschi, and Satoshi Tadokoro, editors, *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Computer Science*, pages 214–223. Springer, 2002. ◇ pp: 129

Peter Stone and Manuela Veloso. Team-partitioned, opaque-transition reinforcement learning. In *RoboCup-98: Robot Soccer World Cup II*, volume 1604 of *Lecture Notes in*

Artificial Intelligence, pages 261–272, 1999. Also in Proceedings of the Third International Conference on Autonomous Agents (Agents-99). ◇ pp: 4, 6, 7, 53, 71, 99, 127

Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. MIT, 1996. ◇ pp: 5

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998. ◇ pp: 45, 81

Katia P. Sycara. Multiagent systems. *AI Magazine*, 10(2):79–92, Summer 1998. ◇ pp: 43

Milind Tambe. Teamwork in real-world, dynamic environments. In Mario Tokoro, editor, *Proceedings of the Second International Conference on Multi-Agent Systems (ICMAS-96)*, pages 361–368, Kyoto, Japan, 1996. AAAI Press. ◇ pp: 129

Milind Tambe. Agent architectures for flexible, practical teamwork. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 22–28, Providence, Rhode Island, USA, July 1997a. ◇ pp: 129

Milind Tambe. Towards flexible teamwork. *Journal of Artificial Intelligence Research*, 7: 83–124, 1997b. ◇ pp: 129

Milind Tambe, Jafar Adibi, Yaser Al-Onaizan, Ali Erdem, Gal A. Kaminka, Stacy C. Marsella, and Ion Muslea. Building agent teams using an explicit teamwork model and learning. *Artificial Intelligence*, 110:215–240, 1999. ◇ pp: 6

Austin Tate. Generating project networks. In Raj Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI 1977)*, pages 888–893. William Kaufmann, August 1977. ISBN 0-86576-057-8. (Reprinted in [Allen et al., 1990], pages 291–296). ◇ pp: 16

Austin Tate, James Hendler, and Mark Drummond. A review of AI planning techniques. In James Allen, James Hendler, and Austin Tate, editors, *Readings in Planning*, pages 26–49. Morgan Kaufmann, 1990. ◇ pp: 16

Costas Tsatsoulis and Andrew B. Williams. Case-based reasoning. In Cornelius T. Leondes, editor, *Knowledge-Based Systems: Techniques and Applications*, volume 3, pages 807–837. Academic Press, 2000. ◇ pp: 81

Iwan Ulrich and Johann Borenstein. VFH+: Reliable obstacle avoidance for fast mobile robots. In *IEEE International Conference on Robotics and Automation (ICRA 1998)*, volume 2, pages 1572–1577, 1998. ◇ pp: 133

Iwan Ulrich and Johann Borenstein. VFH*: Local obstacle avoidance with look-ahead verification. In *IEEE International Conference on Robotics and Automation (ICRA 2000)*, volume 3, pages 2505–2511, 2000. ◇ pp: 133

Steven A. Vere. Planning in time: Windows and durations for activities and goals. *IEEE Transactions On Pattern Analysis And Machine Intelligence*, 5(3):246–267, May 1983. ◇ pp: 13

Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8:279–292, 1992. ◇ pp: 45

Barry Brian Werger. Cooperation without deliberation: A minimal behavior-based approach to multi-robot teams. *Artificial Intelligence*, 110(2):293–320, 1999. ◇ pp: 6

James Westendorp, Paul Scerri, and Lawrence Cavedon. Strategic behavior-based reasoning with dynamic, partial information. In *Selected Papers from the 11th Australian Joint Conference on Artificial Intelligence (AI 1998)*, volume 1502 of *Lecture Notes in Computer Science*, pages 297–308. Springer, 1998. ◇ pp: 87

Shimon Whiteson and Peter Stone. Concurrent layered learning. In *Proceedings of Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-03)*, page to appear, Melbourne, Australia, 2003. ◇ pp: 5, 55

Marco Wiering, Rafal Salustowicz, and Jürgen Schmidhuber. Reinforcement learning soccer teams with incomplete world models. *Autonomous Robots*, 7(1):77–88, 1999. ◇ pp: 56

David E. Wilkins. Domain-independent planning: Representation and plan generation. *Artificial Intelligence*, 22(3):269–301, 1984. ◇ pp: 13, 14, 16, 78

David E. Wilkins, Karen L. Myers, John D. Lowrance, and Leonard P. Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI*, 7(1):197–227, 1995. ◇ pp: 29

Michael Winer and Karen Ray. *Collaboration Handbook*. Amherst H. Wilder Foundation, 1994. . ◇ pp: 36, 37, 38

Michael Wooldridge and Nicholas R. Jennings. Intelligent Agents: Theory and Practice. *Knowledge Engineering Review*, 10(2):115–152, 1995. ◇ pp: 133

Jeremy Wyatt. Reinforcement learning: A brief overview. In Reiner Kühn, Randolf Menzel, Wolfram Menzel, Ulrich Ratsch, Michael M. Richter, and Ion-Olimpiu Stamatescu, editors, *Adaptivity and Learning: An Interdisciplinary Debate*, pages 243–264. Springer, 2003. ◇ pp: 45

Shlomo Zilberstein and Stuart J. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI 1993)*, pages 1402–1407, Chambéry, France, 1993. ◇ pp: 40

◇

Appendices

APPENDIX A

Test Plans

This appendix lists the four plans we used in our experiments. These plans are in skeletal form. That is, they only contain the high-level descriptions that a plan designer would include before any training takes place.

A.1 Centerpass Plan

```
(plan Centerpass
   (step 1
      ;;------------------------------------------------------------------
      ;; Player A dribbles the ball towards the opponent’s goal line.
      ;; Player B runs to closer to the goal
      ;;------------------------------------------------------------------
      (precondition
         (timeout 40)
         (role A -1 ;; view angle=90
            (has-ball A)
            (in-rectangle-abs B 65 15 20 20)
         )  ;;
         (role B -1 ;; view angle=296.565
            (not has-ball B)
            (in-rectangle-abs A 75 0 20 20)
         )  ;;
      )
      (postcondition
         (timeout 50)
         (role A -1 ;; view angle=90
            (in-rectangle-abs A 95 0 10 20)
            (has-ball A)
         )  ;;
         (role B -1 ;; view angle=296.565
            (in-rectangle-abs B 86.25 20 17.5 30)
         )  ;;
      )
      (application-knowledge)
   )  ;; end step

   (step 2
      ;;--------------------------------------------------------------------
      ;; Player A passes the ball to player C
      ;;--------------------------------------------------------------------
      (precondition
         (timeout 30)
         (role A -1 ;; view angle=90
            (in-rectangle-abs B 86.25 20 17.5 30)
         )  ;;
         (role B -1 ;; view angle=296.565
            (true B)
         )  ;;
      )
      (postcondition
         (timeout 60)
         (role A -1
            (has-ball B)
         )  ;;
         (role B -1
            (ready-to-receive-pass B)
            (has-ball B)
         )
      )
      (application-knowledge)
   )  ;; end step
)  ;; end of Centerpass plan
```

A.2 Givengo Plan

```
(plan Givengo
   (rotation-limit 120 15)
   (step 10
      ;;-----------------------------------------------------------------------
      ;; A passes the ball to B
      ;;-----------------------------------------------------------------------
      (precondition
         (timeout 50)
         (role A -1 ;; view angle=56.3099
            (has-ball A)
            (in-rectangle-rel B 5.96 -2.63 8.74 -6.8 12.9 -4.02 10.12 0.14)
         )  ;;
         (role B -1 ;; view angle=225
            (not has-ball B)
            (in-rectangle-rel A 13.44 -1.41 9.9 2.12 6.36 -1.41 9.9 -4.95)
         )  ;;
      )  ;; end precondition
      (postcondition
         (timeout 50)
         (role A -1 ;; view angle=56.3099
            (has-ball B)
         )  ;;
         (role B -1 ;; view angle=225
            (ready-to-receive-pass B)
            (has-ball B)
         )  ;;
      )  ;; end postcondition
      (application-knowledge)
   )  ;  end step 1
   (step 20
      ;;-----------------------------------------------------------------------
      ;; A runs to its new location
      ;;-----------------------------------------------------------------------
      (precondition
         (timeout 15)
         (role A -1 ;; view angle=56.3099
            (has-ball B)
         )  ;;
         (role B -1 ;; view angle=225
            (has-ball B)
         )  ;;
      )  ;; end step 2
      (postcondition
         (timeout 35)
         (role A -1 ;; view angle=56.3099
            (in-rectangle-rel A 3.74 7.9 6.52 3.74 10.68 6.52 7.9 10.68)
         )  ;;
         (role B -1 ;; view angle=225
            (in-rectangle-rel A 7.78 -9.9 4.24 -6.36 0.71 -9.9 4.24 -13.44)
         )  ;;
      )  ;; end postcondition
      (application-knowledge)
   ) ;;  end step 1
   (step 30
      ;;-------------------------------------------------------------------
      ;; B passes the ball back to A
      ;;-------------------------------------------------------------------
      (precondition
         (timeout 25)
         (role A -1 ;; view angle=56.3099
            (has-ball B)
         )  ;;
         (role B -1 ;; view angle=225
            (has-ball B)
         )  ;;
      )  ;; end preconditions
      (postcondition
         (timeout 30)
         (role A -1 ;; view angle=56.3099
            (ready-to-receive-pass A)
            (has-ball A)
         )  ;;
         (role B -1 ;; view angle=225
            (has-ball A)
         )  ;;
      )  ;; end postconditions
      (application-knowledge)
   ) ;;  end step 3
)  ;  end of Givengo plan
```

A.3 Singlepass Plan

```
(plan Singlepass
   (rotation-limit 120 15)
   (step 10
      ;;----------------------------------------------------------------------
      ;; A passes the ball to B
      ;;----------------------------------------------------------------------
      (precondition
         (timeout 15)
         (role A 10 ;; view angle=90
            (has-ball A)
            (in-rectangle-rel B 12.5 2.5 12.5 -2.5 17.5 -2.5 17.5 2.5)
         )  ;;
         (role B -1 ;; view angle=270
            (not has-ball B)
            (in-rectangle-rel A 17.5 -2.5 17.5 2.5 12.5 2.5 12.5 -2.5)
         )  ;;
      )  ;; end precondition
      (postcondition
         (timeout 40)
         (role A -1
            (has-ball B)
         )  ;;
         (role B -1
            (ready-to-receive-pass B)
            (has-ball B)
         )  ;;
      )  ;; end postcondition
   )  ;  end step 1
)  ;  end of Singlepass plan
```

A.4 UEFA Fourwaypass Plan

```
(plan UEFA_Fourwaypass
   (rotation-limit 120 30)
   (step 1
      ;;-----------------------------------------------------------------------
      ;; Player B passes the ball to player A
      ;;-----------------------------------------------------------------------
      (precondition
         (timeout 40)
         (role A -1 ;; view angle=45
            (not has-ball A)
            (in-rectangle-rel B 3.54 -10.61 10.61 -17.68 17.68 -10.61 10.61 -3.54)
            (in-rectangle-rel C 14.14 0 21.21 -7.07 28.28 0 21.21 7.07)
         )  ;;
         (role B -1 ;; view angle=135
            (has-ball B)
            (not has-ball C)
            (in-rectangle-rel A 10.61 17.68 3.54 10.61 10.61 3.54 17.68 10.61)
            (in-rectangle-rel C 10.61 -3.54 3.54 -10.61 10.61 -17.68 17.68 -10.61)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball B)
            (in-rectangle-rel A 28.28 0 21.21 7.07 14.14 0 21.21 -7.07)
            (in-rectangle-rel B 17.68 10.61 10.61 17.68 3.54 10.61 10.61 3.54)
         )  ;;
      )
      (postcondition
         (timeout 50)
         (role A -1 ;; view angle=45
            (ready-to-receive-pass A)
            (has-ball A)
         )  ;;
         (role B -1 ;; view angle=135
            (has-ball A)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball A)
         )  ;;
      )
      (application-knowledge)
   )  ;; end step
   (step 2
      ;;-----------------------------------------------------------------------
      ;; Player A passes the ball to player C
      ;;-----------------------------------------------------------------------
      (precondition
         (timeout 40)
         (role A -1 ;; view angle=45
            (has-ball A)
            (in-rectangle-rel C 14.14 0 21.21 -7.07 28.28 0 21.21 7.07)
         )  ;;
         (role B -1 ;; view angle=135
            (true B)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball A)
            (in-rectangle-rel A 28.28 0 21.21 7.07 14.14 0 21.21 -7.07)
         )  ;;
      )
      (postcondition
         (timeout 50)
         (role A -1 ;; view angle=45
            (has-ball C)
         )  ;;
         (role B -1
            (true B)
         )
         (role C -1 ;; view angle=225
            (ready-to-receive-pass C)
            (has-ball C)
         )  ;;
      )
      (application-knowledge)
   )  ;; end step
   (step 3
      ;;-------------------------------------------------------------------------
      ;; Player A runs to its new location
      ;;-------------------------------------------------------------------------
      (precondition
         (timeout 25)
         (role A -1 ;; view angle=45
            (has-ball C)
         )  ;;
         (role B -1 ;; view angle=135
            ;;(has-ball C)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball C)
         )  ;;
      )
      (postcondition
         (timeout 60)
         (role A -1 ;; view angle=45
            ;;(has-ball C)
            (in-rectangle-rel C 14.14 0 21.21 -7.07 28.28 0 21.21 7.07)
```

```
            (in-rectangle-rel A 3.54 10.61 10.61 3.54 17.68 10.61 10.61 17.68)
         )  ;;
         (role B -1 ;; view angle=135
            (has-ball C)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball C)
            (in-rectangle-rel A 17.68 -10.61 10.61 -3.54 3.54 -10.61 10.61 -17.68)
         )  ;;
      )
   )
   (step 4
      ;;-------------------------------------------------------------------------
      ;; Player C passes the ball back to A
      ;;-------------------------------------------------------------------------
      (precondition
         (timeout 25)
         (role A -1 ;; view angle=45
            ;;(has-ball C)
         )  ;;
         (role B -1 ;; view angle=135
            ;;(has-ball C)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball C)
         )  ;;
      )
      (postcondition
         (timeout 60)
         (role A -1 ;; view angle=45
            (ready-to-receive-pass A)
            (has-ball A)
         )  ;;
         (role B -1 ;; view angle=135
            (has-ball A)
         )  ;;
         (role C -1 ;; view angle=225
            (has-ball A)
         )  ;;
      )
   )  ;; end step
)  ;; end of UEFA_Fourwaypass plan
```

A.5 Interceptball Plan

```
(plan Interceptball
   (step 1
      (precondition
         (role A -1
            (true A)
         )  ;; end role A
      )  ;; end precondition
      (postcondition
         (timeout 50)
         (role A -1
            (has-ball A)
         )  ;; end role A
      )  ;; end postcondition
      (application-knowledge
         (case Interceptball A 1
            (action-sequence
               (intercept-ball A)
            )  ;; end action-sequence
         )  ;; end case
      )  ;; end application-knowledge
   )  ;; end step 1
   (step 2
      (goto 1)
    )
)  ;; end plan Interceptball
```

◇

APPENDIX **B**

Example Plan

This appendix lists parts of the Givengo plan (*Section A.2*) after training and learning. Due to the actual length of this plan after training, we only include some example cases for each step.

```
(plan Givengo
   (success 685)
   (failure 381)
   (rotation-limit 120 15)
   (step 10
      (precondition
         (timeout 50)
         (role A -1
            (has-ball A)
            (in-rectangle-rel B 5.96 -2.63 8.74 -6.8 12.9 -4.02 10.12 0.14)
         )  ;; end role A
         (role B -1
            (not has-ball B)
            (in-rectangle-rel A 13.44 -1.41 9.9 2.12 6.36 -1.41 9.9 -4.95)
         )  ;; end role B
      )  ;; end precondition
      (postcondition
         (timeout 50)
         (role A -1
            (has-ball B)
         )  ;; end role A
         (role B -1
            (has-ball B)
            (ready-to-receive-pass B)
         )  ;; end role B
      )  ;; end postcondition
```

```
(application-knowledge
   (case Givengo A 10
      (grid-unit 1.5)
      (success 753)
      (failure 8)
      (rotation 0)
      (opponent-constellation
      )
      (action-sequence
         (pass-ball A B)
      )  ;; end action-sequence
   )  ;; end case
   (case Givengo A 10
      (grid-unit 1.5)
      (success 23)
      (failure 15)
      (rotation 0)
      (opponent-constellation
         (7 -2)
      )
      (action-sequence
         (dribble-ball-rel A 6.75 0.75 6.75 -2.25)
         (pass-ball A B)
      )  ;; end action-sequence
   )  ;; end case
   (case Givengo A 10
      (grid-unit 1.5)
      (success 3)
      (failure 9)
      (rotation 0)
      (opponent-constellation
         (4 -3)
         (7 -1)
      )
      (action-sequence
         (dribble-ball-rel A 3.75 0.75)
         (pass-ball A B)
      )  ;; end action-sequence
   )  ;; end case
   (case Givengo A 10
      (grid-unit 1.5)
      (success 8)
      (failure 12)
      (rotation -15)
      (opponent-constellation
         (5 -1)
      )
      (action-sequence
            (dribble-ball-rel A 2.25 -0.75 2.25 -2.25 3.75 -2.25
                                3.75 -3.75 5.25 -3.75 5.25 -6.75)
            (pass-ball A B)
         )  ;; end action-sequence
      )  ;; end case
      (case Givengo A 10
         (grid-unit 1.5)
         (success 1)
         (failure 2)
         (rotation 10.4218)
         (opponent-constellation
            (7 1)
            (5 -2)
            (2 0)
         )
         (action-sequence
            (dribble-ball-rel A 6.75 0.75)
            (pass-ball A B)
         )  ;; end action-sequence
      )  ;; end case
      (case Givengo A 10
         (grid-unit 1.5)
         (success 1)
         (failure 0)
         (rotation -69.4121)
         (opponent-constellation
            (0 5)
            (1 1)
            (0 1)
            (-1 0)
         )
         (action-sequence
            (dribble-ball-rel A 0.75 -2.25)
            (pass-ball A B)
         )  ;; end action-sequence
      )  ;; end case

      . . .

   )  ;; end application-knowledge
)  ;; end step 10
(step 20
   (precondition
      (timeout 15)
      (role A -1
         (has-ball B)
      )  ;; end role A
      (role B -1
      (has-ball B)
   )  ;; end role B
)  ;; end precondition
(postcondition
   (timeout 35)
   (role A -1
      (in-rectangle-rel A 3.74 7.9 6.52 3.74 10.68 6.52 7.9 10.68)
   )  ;; end role A
   (role B -1
      (in-rectangle-rel A 7.78 -9.9 4.24 -6.36 0.71 -9.9 4.24 -13.44)
   )  ;; end role B
)  ;; end postcondition
(application-knowledge
   (case Givengo A 20
      (grid-unit 1.5)
      (success 7)
      (failure 2)
      (rotation -69.3622)
      (opponent-constellation
      )
      (action-sequence
         (goto-area-rel A 8.71123 -0.715564 5.79803 -4.78339
                          9.86586 -7.6966 12.7791 -3.62877)
      )  ;; end action-sequence
   )  ;; end case
   (case Givengo A 20
      (grid-unit 1.5)
      (success 1)
      (failure 1)
      (rotation -60)
      (opponent-constellation
         (7 3)
      )
      (action-sequence
         (goto-area-rel A 8.7116 0.711065 6.49894 -3.77649
                          10.9865 -5.98915 13.1992 -1.5016)
      )  ;; end action-sequence
   )  ;; end case
   (case Givengo A 20
      (grid-unit 1.5)
      (success 3)
      (failure 0)
      (rotation 10.4218)
      (opponent-constellation
         (7 2)
         (4 -1)
      )
      (action-sequence
            (goto-area-rel A 2.24925 8.44621 5.7359 4.85772
                             9.32439 8.34437 5.83774 11.9329)
         )  ;; end action-sequence
      )  ;; end case
      (case Givengo A 20
         (grid-unit 1.5)
         (success 1)
         (failure 0)
         (rotation -3.57823)
         (opponent-constellation
            (5 2)
            (7 1)
            (-1 -1)
         )
         (action-sequence
            (goto-area-rel A 4.22576 7.65118 6.74071 3.32579
                             11.0661 5.84074 8.55115 10.1661)
         )  ;; end action-sequence
      )  ;; end case
      (case Givengo A 20
         (grid-unit 1.5)
         (success 1)
         (failure 0)
         (rotation -24.5782)
         (opponent-constellation
            (2 3)
            (2 5)
            (8 3)
            (3 0)
            (0 -1)
         )
         (action-sequence
            (goto-area-rel A 6.68702 5.62862 7.48485 0.689236
                             12.4242 1.48706 11.6264 6.42644)
         )  ;; end action-sequence
      )  ;; end case
   )  ;; end application-knowledge

   • • •

)  ;; end step 20
(step 30
   (precondition
      (timeout 25)
      (role A -1
         (has-ball B)
      )  ;; end role A
      (role B -1
            (has-ball B)
         )  ;; end role B
      )  ;; end precondition
      (postcondition
         (timeout 30)
         (role A -1
            (has-ball A)
            (ready-to-receive-pass A)
         )  ;; end role A
         (role B -1
            (has-ball A)
         )  ;; end role B
      )  ;; end postcondition
      (application-knowledge
         ;; similarity=0
         (case Givengo A 30
            (grid-unit 1.5)
            (success 686)
            (failure 381)
            (rotation 0)
            (opponent-constellation
            )
            (action-sequence
               (intercept-ball A)
            )  ;; end action-sequence
         )  ;; end case
      )  ;; end application-knowledge
   )  ;; end step 30
)  ;; end plan Givengo
```

◇

APPENDIX C

Synchronization of clients with Soccer Server

A client program can be synchronized with the Soccer Server using an undocumented feature of the Server that has to do with how the Soccer Server handles the switching of the viewing modes on behalf of client programs. By taking advantage of the deterministic behavior of the Server in response to viewing mode changing commands from clients, it is possible for each client program to schedule incoming visual feedback messages from the Server such that a client program can have the longest possible time during an action cycle to reason about its next action. [1]

The Server provides visual feedback period to each client program once every 150ms, and each client can send action commands to the Server once every 100ms. This means that a client receives two visual feedbacks out of every consecutive three action cycles. If a client changes its viewing mode to *narrow* **or** *low*, the visual feedback period reduces to one-half of the default, i.e., 75ms. If the client changes its view mode to *narrow* **and** *low*, the feedback period reduces to one-fourth of the default, i.e., 37.5ms. This means that the server checks every 37.5ms to see if it is time to send a visual feedback to this client. Therefore, assuming no other delays, the server will check whether to send a visual to the client or not at intervals 0ms, 37.5ms, 75ms, 112.5ms, 150ms, 187.5ms, 225ms, 262.5ms and

[1] This appendix is an edited and supplemented version of an email message sent by Tom Howard (`thoward@tibco.com`) on March 5, 2002, to the RoboCup simulation league mailing list (`robocup-sim-l@usc.edu`) describing how to synchronize client programs with the Soccer Server.

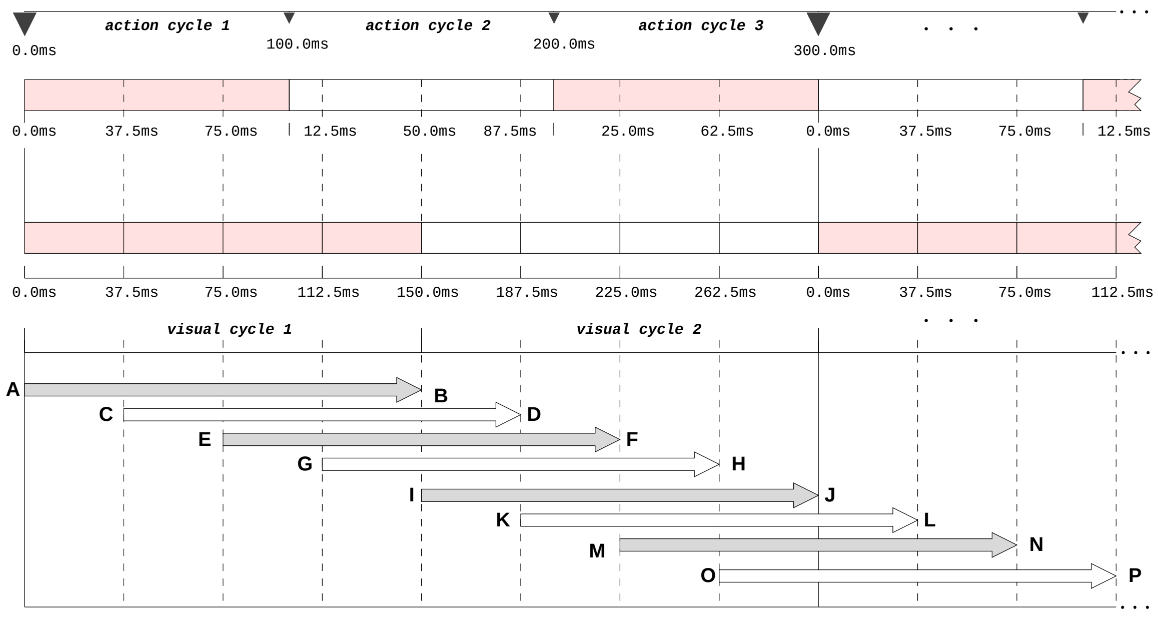


Figure C.1: Timing of visual feedback messages sent by the server with respect to action cycles

300ms every 3 action cycles (equivalently, every 2 visual cycles). At the 300ms mark, the entire cycle sequence completes and a new one starts. That is, the feedback cycle is back to the 0ms point. So, for example, if the server sent a visual feedback at the 37.5ms mark, it would send another 150ms later, at the 187.5ms mark. The next feedback would be sent at the 37.5ms mark of the first action cycle in the next cycle sequence.

Figure C.1 is a diagram that depicts all decision points at which the server may check whether it needs to send a visual feedback to a given client. On top of the diagram are action cycles 1, 2, 3, and so on. In two of every three consecutive action cycles, each client receives a visual feedback. This is why the timeline in the middle part of the diagram shows two visual cycles. The bottom timeline shows all possible pairs of decision points at which the server may send visual feedback messages to a given client.

As *Figure C.1* shows, the intervals 0ms, 37.5ms, 75ms, 112.5ms, 150ms, 187.5ms, 225ms, 262.5ms and 300ms, map onto eight different intervals in a single action cycle. Since the server goes through the same procedure every cycle, it has to check whether to send a visual feedback to a client at 0ms, 12.5ms, 25ms, 37.5ms, 50ms, 62.5ms, 75ms or 87.5ms.

The mapping of the 300ms interval marks onto single action cycle intervals marks is given in *Table C.1*. Note that the order of the values on the right-hand-side in *Table C.1* is unordered as opposed to the values given above.

Visual feedback times in a 300ms period		Visual feedback times in a single (100ms) action cycle
0.0ms	$\longrightarrow$	0.0ms
37.5ms	$\longrightarrow$	37.5ms
75.0ms	$\longrightarrow$	75.0ms
112.5ms	$\longrightarrow$	12.5ms
150.0ms	$\longrightarrow$	50.0ms
187.5ms	$\longrightarrow$	87.5ms
225.0ms	$\longrightarrow$	25.0ms
262.5ms	$\longrightarrow$	62.5ms
300.0ms	$\longrightarrow$	0.0ms

Table C.1: Mapping of all possible visual feedback message times in a 300ms period onto a single action cycle

So, if a client is in default viewing mode, the server will send visual feedback to it every 150ms. However, there is no facility to specify whether this will be done at the beginning of one cycle, the middle of the next, or any other time. So, depending on when the client happens to connect to the server during a simulation cycle, the client will get visual feedback at one of the four intervals in the two consecutive action cycles shown in *Table C.2*:

Action Cycle T	Action Cycle $(T+1)$	Timeline	Reverse Timeline
0ms	50ms	A–B	I–J
12.5ms	62.5ms	G–H	O–P
25ms	75ms	M–N	E–F
37.5ms	87.5ms	C–D	K–L

Table C.2: The visual feedback time offsets from the beginning of an action cycle and corresponding timelines shown in *Figure C.1*

For example, if the client receives visual feedback right at the beginning of action cycle T, it will receive the next visual feedback after 50ms from the beginning of the next action cycle, $(T+1)$. If it receives visual feedback after 12.5ms from the beginning of T, then it will receive its next feedback 62.5ms after the start of action cycle $(T+1)$, and so on.

The "Timeline" column in *Table C.2* shows the timeline in *Figure C.1* that each pair of values corresponds to. The ”Reverse Timing” column is equivalent to the ”Timeline” column, except that the order of the timing values are reversed, since the process is cyclic or symmetric. For example timing line G–H starts at 12.5ms and extends until 62.5ms. Its reverse version, O–P, starts at 62.5ms and extends until the 12.5ms point in the next cycle.

We can see from *Table C.2* that the first pair is the most desirable because it has the least average offset from the beginning of the action cycles during which the client will receive visual feedback. That is, we wish to get visual feedback messages from the server as early as we can in any given cycle so that the reasoning time for an agent can be maximized. Conversely, the last pair is the least desirable, since it has the highest average offset duration.

Since when a client connects to the server cannot be controlled precisely, the visual feedback timing intervals of the client will match one of the above possibilities. However, by taking advantage of the server behavior, it is possible to modify the timings after a client connects to the server so that the client can select the precise timing of the future visual feedbacks it will receive from the server (assuming zero delay in message transmission from the server to the client. The times shown are therefore those at which the server can potentially send visual feedback messages.)

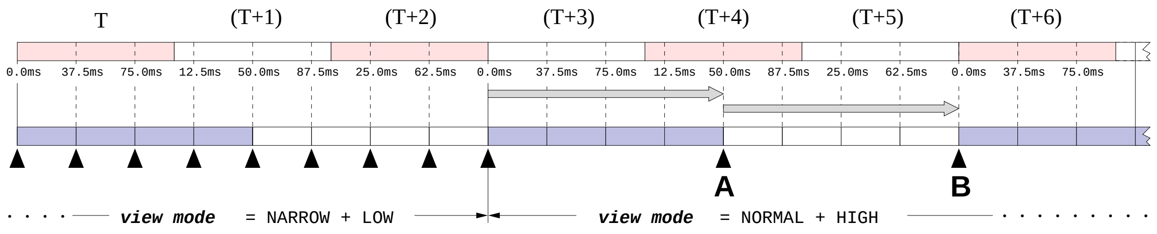


Figure C.2: A possible synchronization process of the client with the server

Figure C.2 demonstrates this synchronization process. If the client changes its viewing mode to *narrow* **and** *low*, it will receive visual feedback every 37.5ms. In the figure, the time at which the server sends a visual feedback message is indicated by an arrowhead.

On close inspection of *Figure C.1*, it will become clear that, in every 3 cycles, the client will receive 3 visual feedbacks per cycle in 2 consecutive action cycles and 2 visual feedbacks per cycle in another single action cycle. So if the client receives 3 visual feedbacks in cycle T, 3 in cycle $(T + 1)$, then we know, with certainty, that the client will receive 2 visual feedbacks in cycle $(T + 2)$.

Moreover, if the client changes its viewing mode to *normal* **and** *high* after the very first visual feedback in $(T + 3)$, which would arrive at the 0ms mark in the overall cycle, then the next visual feedback in cycle $(T + 4)$ will be at 50ms from the beginning of that cycle (see point A in *Figure C.2*), and the one following visual feedback will arrive in cycle $(T+6)$ at 0ms (see point B in *Figure C.2*) and so on. From this point on, the client starts receiving visual feedbacks as early as possible in each cycle, that is at 0ms and 50ms offset points.

We should also note that the client will need to do the same trick after every change to *narrow* **and** *low* mode, and it will need to do a similar trick for after every change to *narrow* **or** *low*.

Previously, we mentioned that the server checks to see if it needs to send visual feedback messages every 37.5ms. The server actually *cannot* send visual feedbacks every 37.5ms, since the internal timer interval in the server is 10ms, which is the lowest possible resolution we can get on most machines. Since 37.5 is not a multiple of 10, some rounding off occurs in the server. *Table C.3* gives the correct visual feedback pairs (compare these values to the ones in *Table C.2*).

Action Cycle T	Action Cycle $(T + 1)$
0ms	50ms
20ms	70ms
30ms	80ms
40ms	90ms

Table C.3: Actual visual feedback time offsets from the beginning of each action cycle

As we can see, rounding off makes bad synchronization even worse, because the server ends up sending its visual feedbacks later (*Table C.3*) than it would have otherwise (*Table C.2*) if higher resolution timer intervals were possible.

◇

APPENDIX D

Position triangulation in Soccer Server

This appendix describes how to triangulate a position given the possibly noisy polar coordinates of two points whose global coordinates are known. We assume that the angle components of the polar coordinates are relative to the current line of sight of the observer where counterclockwise angles are negative and clockwise angles are positive. For example, in *Figure D.1*, if we suppose that the observer is at point I_0 and that it sees two objects located at points A and B, then the polar coordinates of A would be given as (r_A, t_A) and the polar coordinates of B would be given as (r_B, t_B), where $t_A = \delta$ and $t_B = \omega$ are angles relative to the observer's current *line of sight* indicated in the figure.

Using the radii values in the two polar coordinate pairs, we can draw two circles, whose centers are at A and B. Both circles pass through points I_0 and I_1 where they intersect. Then line segment $\overline{\mathrm{AB}}$ is then, by definition, perpendicular to the line segment $\overline{\mathrm{I_0I_1}}$ whose middle point is at M. Also, we have the following relationships

$$\overline{\mathrm{I_0A}} = \overline{\mathrm{I_1A}} = r_A,$$

$$\overline{\mathrm{I_0B}} = \overline{\mathrm{I_1B}} = r_B,$$

$$\overline{\mathrm{I_1M}} = \overline{\mathrm{I_0M}} = h,$$

$$\overline{\mathrm{AM}} = a, \overline{\mathrm{MB}} = b, \text{and}$$

$$d = a + b$$

where values r_A, r_B, and d will be given in the triangulation problem.

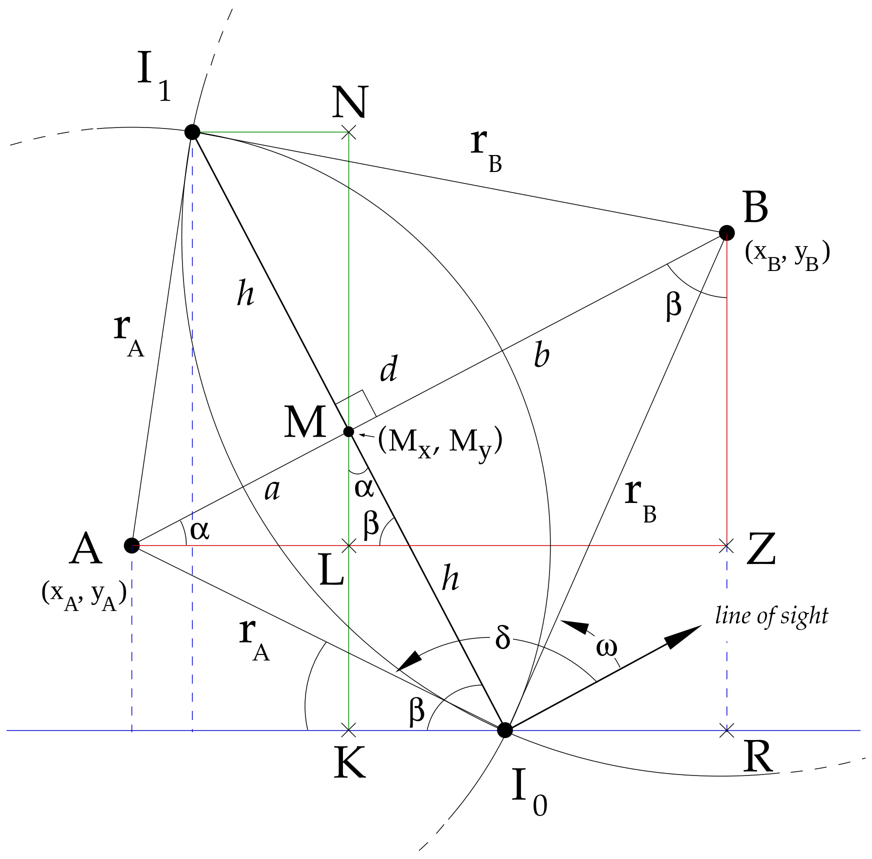


Figure D.1: The setup of the position triangulation problem that involves determining the global position of the observer and the global angle of its line of sight, given polar coordinates of two points A and B relative to the observer, (r_A, δ) and (r_B, ω). The global (x, y) coordinates of A and B are assumed to be known. Two circles drawn centered at A and B with radii r_A and r_B intersect at I_0 and I_1. Therefore, the observer is at one of these two intersection points. By definition, the line that joins the two circle centers, A and B, cuts the line that joins the two intersection points I_0 and I_1 into two halves at M, (M_x, M_y), each half of length h. It is also the case that the distance between A and either of the intersection points is equal to the polar radius r_A. Similarly, the distance from B to either of the intersection points is r_B. $|MA| = a$, $|MB| = b$, and $d = a + b$.

In summarized form, the triangulation process we implemented works as follows:

1. First, we compute the (x, y) coordinates of M, (M_x, M_y).

2. Then using (M_x, M_y), we compute the (x, y) coordinates of I_0 and I_1.

3. Finally, we test the given relative angles δ and ω to decide whether the observer is at I_0 or I_1.

To help illustrate the triangulation process, we have already supposed that the observer is at I_0, but note that this knowledge about the location of the observer will not be available in the problem. Instead, we will have two possible points to choose from, namely I_0 and I_1 as shown in *Figure D.1*. We must also note that *Figure D.1* only illustrates the most general case where there are two intersections between the two circles formed by the given polar coordinates of the two known objects. It is also possible that the two circles may intersect tangentially, that is, at one single point. In this case, h would be 0 and $a = r_A$ and $b = r_B$.

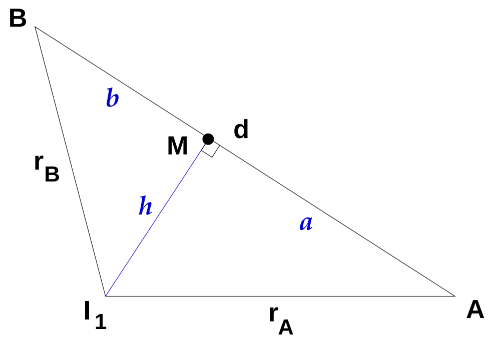


Figure D.2: $\triangle I_1AB$ from *Figure D.1* with height h, $d = b + a$, $|\mathrm{BM}| = b$ and $|\mathrm{MA}| = a$

To compute the coordinates of M, we have to compute the length of line segment $\overline{\mathrm{KI}_0}$ (or $\overline{\mathrm{NI}_1}$). Since, in triangle $\mathrm{AI_1B}$, we know r_A, r_B, and d, we can compute h, a, and b.

The computation of d is straightforward. Since the global coordinates of A (x_A, y_A) and B (x_B, y_B) are known, we have:

$$d = \sqrt{({x_A}^2 - {x_B}^2) + ({y_A}^2 - {y_B}^2)}$$

To see how we can compute a and b, let us look at *Figure D.2* that shows the same scenario where $\triangle I_1AB$ is a triangle with height h. For this triangle, we can compute b, a, and h, given d, r_B, and r_A. Based on triangle $\triangle I_1AB$, we have the following relationships:

$$b^2 + h^2 = r_B^2 \quad \text{(D.1)}$$

$$a^2 + h^2 = r_A^2 \quad \text{(D.2)}$$

Subtracting *Equation D.2* from *Equation D.1*, we get:

$$\begin{array}{rrcl} & b^2 + h^2 & = & r_B^2 \\ - & a^2 + h^2 & = & r_A^2 \\ \hline & b^2 - a^2 & = & r_B^2 - r_A^2 \end{array}$$

Rearranging the left-hand size of the last equation above for a^2, we get

$$a^2 = r_A^2 + b^2 - r_B^2 \tag{D.3}$$

Since $b + a = d$, then

$$b = d - a \tag{D.4}$$

Substituting for b from *Equation D.4* into *Equation D.3*, we get:

$$\begin{aligned} a^2 &= r_A^2 + (d-a)^2 - r_B^2 \\ a^2 &= r_A^2 + d^2 - 2da + a^2 - r_B^2 \\ 2da &= r_A^2 + d^2 - r_B^2 \end{aligned} \tag{D.5}$$

Dividing both sides by $2d$ in *Equation D.5*, we get the equation for a:

$$\boxed{a = \frac{r_A^2 - r_B^2 + d^2}{2d}} \tag{D.6}$$

To get the equation for b, we substitute for a from *Equation D.6* into *Equation D.4* and get:

$$b = d - \frac{r_A^2 - r_B^2 + d^2}{2d}$$

Multiplying both sides with $2d$, we get:

$$2db = 2d^2 - r_A^2 + r_B^2 - d^2$$

$$2db = d^2 - r_A^2 + r_B^2$$

Then

$$\boxed{b = \frac{r_B^2 - r_A^2 + d^2}{2d}} \tag{D.7}$$

Then, in summary, for *Figure D.1*, we have

$$a = \frac{{r_A}^2 - {r_B}^2 + d^2}{2d} \tag{D.8}$$

$$b = \frac{{r_B}^2 - {r_A}^2 + d^2}{2d} \tag{D.9}$$

$$h = \sqrt{{r_A}^2 - a^2} \tag{D.10}$$

$$h = \sqrt{{r_B}^2 - b^2} \tag{D.11}$$

Before proceeding further, we must first check whether an intersection is possible between the circles formed by the radii of A and B. For intersection to occur, the distance between the two circle centers must be at least as large as the addition of the two radii. So, we check the truth value of the condition $((d - (r_A + r_B)) > 0)$. If true, it means there can be no intersection between the two circles. This will be the case if there is inconsistency in the global position data or the reported radii values, and, due to this inconsistency, the triangulation computation aborts at this point. If the above condition is false, we proceed by checking if the computed h value is consistent with the problem. We know from *Equation D.10* and *Equation D.11* that $|a|$ cannot be larger than r_A and $|b|$ cannot be larger than r_B. At these border conditions, $|a| = r_A$ and $|b| = r_B$. For example, consider the situation in *Figure D.3*.

In general, if the input data is noisy, we can have situations where h becomes a complex number, which cannot possibly be the case in reality. So, in such cases, we have to set h to zero, which is the smallest value h can have. This correction is required when, for example, the r_A value reported is smaller than it actually is. Consider, for example, the following input data values for the triangulation problem in *Figure D.3*

$$\begin{aligned} r_A &= \sqrt{100}, \quad (x_A, y_A) = (10, 2), \\ r_B &= \sqrt{234}, \quad (x_B, y_B) = (15, 3), \\ t_A &= 0, \qquad t_B = 0 \end{aligned}$$

where the r_A value is less than its actual $\sqrt{104}$. If the r_A value were noiseless, as is the case with circles drawn with a solid line in *Figure D.3*, there would be a single point of

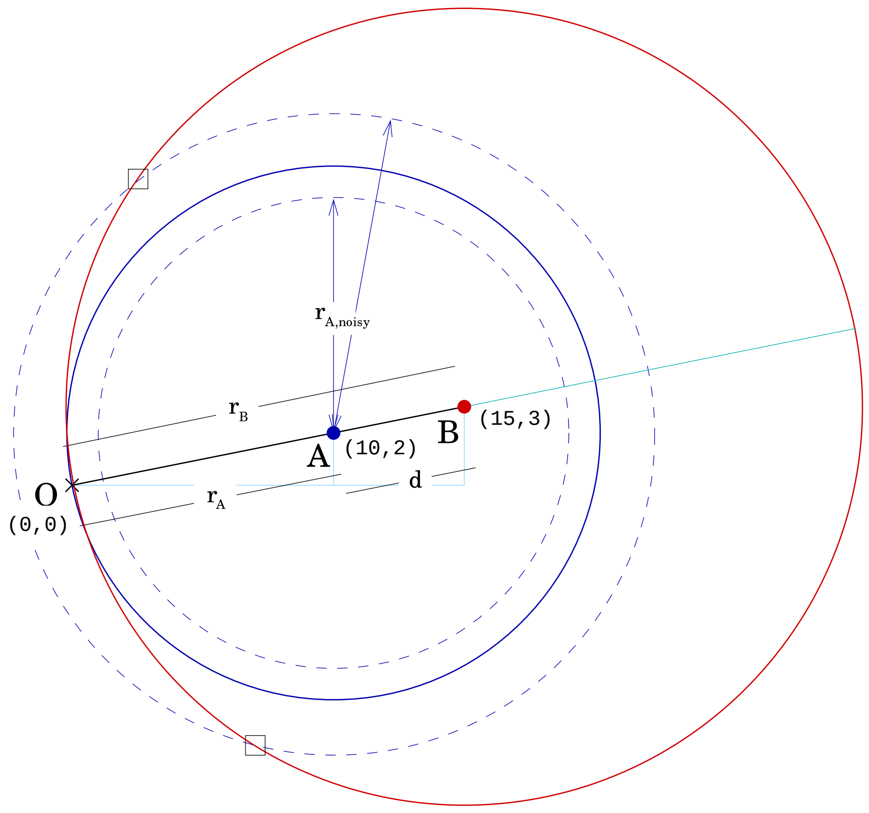


Figure D.3: An example case of triangulation where the known reference points and the position of the observer are collinear. The solid circles depict the case with no noise, and the dashed circle associated with point A depicts the case when r_A is reported less than its actual value. The observer, O, is at $(0,0)$, A is at $(10,2)$, and B is at $(15,3)$. The circles associated with two noisy versions of the r_A value ($r_{A,\text{noisy}}$) are drawn with a dashed line.

intersection as shown. Since r_A value is a smaller than its true value, we can see that that the two circles would not intersect, not because they are far apart from each other but because one is completely enclosed in the other. Based on the above input values, using *Equation D.8*, we get:

$$a = \frac{(\sqrt{100})^2 - (\sqrt{234})^2 + (\sqrt{26})^2}{2 * \sqrt{26}} = -10.59$$

Using *Equation D.10*, we get $h = \sqrt{100 - (-10.59^2)} = \sqrt{-12.15}$, hence the imaginary value we get for h. If $r_A < |a|$, the term under the square-root becomes negative, and this leads to imaginary h values. Since the smallest value of h is 0, then the largest value of a is r_A, i.e., $0 \leq |a| \leq r_A$. That is, $|a|$ can, in reality, never be larger than r_A. Therefore, to correct this situation that may arise due to noise in the input data, we set h to zero. The reason for the a value above being negative is due to the height segment not intersecting the base of the triangle between the two points, A and B, that is, strictly within the $\overline{\mathrm{AB}}$ segment. If we take a mirror image of A on the opposite side of O, this time a would be the mirror of the previous value but positive: $+10.59$. In addition, where the height segment intersects the base (point O) would be within the $\overline{\mathrm{AB}}$ segment –d would naturally have a different value also. Since the a value is always squared when used, the end result does not change.

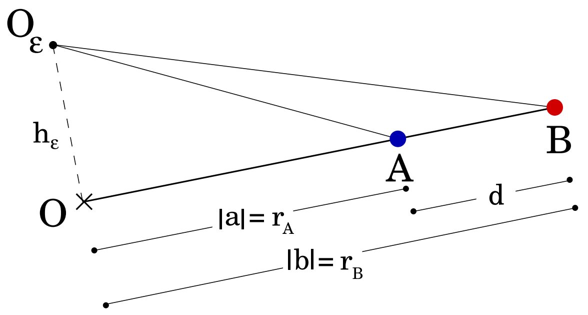


Figure D.4: Computation of the two base segments (a and b) in a triangle formed by the height segment intersecting the base when the height is zero. The figure shows an infinitesimal height segment, h_ϵ to demonstrate that when the height segment does not intersect the base of the triangle between the two vertices that form the base (A and B), the value of a will be negative by *Equation D.8*. By *Equation D.10* and *Equation D.11*, $|a| = r_A$ and $|b| = r_B$

Next, we compute (M_x, M_y). In *Figure D.1*, triangles $\triangle BAZ$ and $\triangle MAL$ are similar. Therefore, we have the relations

$$\frac{|AZ|}{|AB|} = \frac{|AL|}{|AM|} \Longrightarrow \frac{(x_B - x_A)}{d} = \frac{|AL|}{a}$$

Therefore, $|AL| = a * (x_B - x_A)/d$. Similarly, from the same similar triangles, $\triangle BAZ$ and $\triangle MAL$, we have the relations

$$\frac{|BZ|}{|AB|} = \frac{|ML|}{|AM|} \Longrightarrow \frac{(y_B - y_A)}{d} = \frac{|ML|}{a}$$

Therefore, $|ML| = a * (y_B - y_A)/d$. That is, the (x, y) coordinates of M, relative to A are $(a * (x_B - x_A)/d, a * (y_B - y_A)/d)$. Hence, we have

$$\begin{aligned} M_x &= x_A + (a * (x_B - x_A)/d \\ M_y &= y_A + (a * (y_B - y_A)/d \end{aligned}$$

Now we need to compute the x- and y-offset to points I_0 and I_1 from M. So we need to compute $|I_0K|$ (= $|I_1N|$) and $|MK|$ (= $|MN|$). Using similar triangles $\triangle BAZ$ and $\triangle I_0MK$, we have

$$\frac{|BZ|}{|AB|} = \frac{|I_0K|}{|MI_0|} \Longrightarrow \frac{(y_B - y_A)}{d} = \frac{|I_0K|}{h}$$

Therefore, $|I_0K| = h * (y_B - y_A)/d$. Similarly, $|MK| = h * (x_B - x_A)/d$. Hence, we have the following four equations:

$$\begin{aligned}
I_{0_x} &= M_x + (h * (y_B - y_A)/d) \\
I_{0_y} &= M_y - (h * (x_B - x_A)/d) \\
I_{1_x} &= M_x - (h * (y_B - y_A)/d) \\
I_{1_y} &= M_y + (h * (x_B - x_A)/)d
\end{aligned}$$

Next, we compute the global angles with respect to the horizontal from each intersection point I_0 and I_1 to the two centers A and B (i.e., locations of known markers). We use a user-defined function atan2r, which is similar to the C function atan2 but takes into account the signs of its arguments for determining the quadrant of the resulting angle and resizes the return value to fit the range $[0.0\ ..\ 360.0]$. The reference point from which these angles need to be calculated are the intersection points, not the two centers.

$$\begin{aligned}
\phi_{0A} &= \angle RI_0A = & \text{atan2r}((y_A - I_{0_y}), (x_A - I_{0_x})) \\
\phi_{0B} &= \angle RI_0B = & \text{atan2r}((y_B - I_{0_y}), (x_B - I_{0_x})) \\
\phi_{1A} &= \angle NI_1A = & \text{atan2r}((y_A - I_{1_y}), (x_A - I_{1_x})) \\
\phi_{1B} &= \angle NI_1B = & \text{atan2r}((y_B - I_{1_y}), (x_B - I_{1_x}))
\end{aligned}$$

Since we have four possible angles, we compute four possible global angles of sight for the agent by adding the given relative angles t_A and t_B to the angles we computed above. Note that t_A and t_B are in Server angle convention, and, therefore, we invert them to convert them to the Cartesian convention we use:

$$\Theta_{0A} = \text{resizeAngle}(\phi_{0A} + (-t_A))$$

$$\Theta_{0B} = \text{resizeAngle}(\phi_{0B} + (-t_B))$$

$$\Theta_{1A} = \text{resizeAngle}(\phi_{1A} + (-t_A))$$

$$\Theta_{1B} = \text{resizeAngle}(\phi_{1B} + (-t_B))$$

Now we have two sets of candidate coordinates for the global angle of the current line of vision of the agent. In general, under noiseless conditions, the angles in one of these two sets would be identical. The only exception to this condition is when there is only one intersection point. In that case, all four angles would be identical with noiseless input data. Next, we compute the differences between each pair of angles associated with the same intersection point.

$$\mathit{angleDiff}_{\Theta_{0A},\Theta_{OB}} = \text{angleDifferenceWithSign}(\Theta_{0A}, \Theta_{OB})$$

$$\mathit{angleDiff}_{\Theta_{1A},\Theta_{1B}} = \text{angleDifferenceWithSign}(\Theta_{1A}, \Theta_{1B})$$

where angleDifferenceWithSign is a function that returns a signed angle where the sign indicates the relative direction of the second angle from the first. For example, $\text{angleDifferenceWithSign}(345, 15) = +30$, because to go from the 345 degrees to 15 degrees, we have to traverse 30 degrees counterclockwise; hence the $+$ sign. On the other hand, $\text{angleDifferenceWithSign}(15, 345) = -30$, because getting to angle 345 from 15 degrees requires traversing 30 degrees clockwise; hence the $-$ sign.

When the input data is noisy, the case where there would otherwise be a single point of intersection may degenerate into a case where there are two *symmetrical* intersection points. This situation is shown in *Figure D.3* where the outer dashed circle intersects with the larger circle drawn in solid line. The intersection points are marked with small boxes. So, both sets of candidate angles would have to be correct, but, due to the orientation of the two intersection points, the angle differences computed above in each case would carry a different sign. Also, the farther A and B are from each other, the larger the separation of the two intersection points would be. Therefore, choosing one point or the other would throw off the agent since it is a virtual guarantee that neither are correct. That is why, in such a case, we need to compute the global angle by taking an average of the two angles. But we cannot compute this average as in $((v1+v2)/2)$. Instead we have to use the angle difference values instead, since adding angles in the 1st (0-90 degrees) and 4th (270-360 degrees) quadrants and then dividing the result by 2 would give an incorrect result. Averaging the differences gives us a value in between the angles of the two intersection points, and this would be the closest to the actual global viewing angle. To check for the symmetrical case, we look for the case where the absolute difference between the two angle differences computed above are almost equal. The threshold value $symmetricalAngleMargin$ is a small-enough value to help identify this symmetrical case.

$$angleDiffDiff = \mathrm{fabs}(\mathrm{fabs}(angleDiff_{\Theta_{0A},\Theta_{0B}}) - \mathrm{fabs}(angleDiff_{\Theta_{1A},\Theta_{1B}}))$$

We consider three cases:

1. The symmetrical case, where $(angleDiffDiff < symmetricalAngleMargin)$ is true, where neither I_0 nor I_1 is the proper location of the agent.
2. The case where $(\mathrm{fabs}(angleDiff_{\Theta_{0A},\Theta_{0B}}) < \mathrm{fabs}(angleDiff_{\Theta_{1A},\Theta_{1B}}))$ is true, which means that intersection point I_0 is the actual position of the agent.
3. The final case where $(\mathrm{fabs}(angleDiff_{\Theta_{0A},\Theta_{0B}}) < \mathrm{fabs}(angleDiff_{\Theta_{1A},\Theta_{1B}}))$ is false, which means that intersection point I_1 is the actual position of the agent.

For case 1, we compute the following:

$$\begin{aligned}
global_x &= (I_{0x} + I_{1x})/2 \\
global_y &= (I_{0y} + I_{1y})/2 \\
angle_A &= \Theta_{0A} + (angleDiff_{\Theta_{0A},\Theta_{0B}}/2) \\
angle_B &= \Theta_{0B} + (angleDiff_{\Theta_{1A},\Theta_{1B}}/2) \\
angleDiff_{A,B} &= \mathrm{angleDifferenceWithSign}(angle_A, angle_B) \\
globalAngle &= \mathrm{resizeAngle}(angle_A + (angleDiff_{A,B}/2))
\end{aligned}$$

where $(global_x, global_y)$ are the global (x, y) coordinates of the agent, and $globalAngle$ is its global viewing angle. For case 2, we compute the following:

$$\begin{aligned} global_x &= I_{0x} \\ global_y &= I_{0y} \\ globalAngle &= \text{resizeAngle}(\Theta_{0A} + (angleDiff_{\Theta_{0A},\Theta_{0B}}/2)) \end{aligned}$$

Finally, for case 3, we compute the following:

$$\begin{aligned} global_x &= I_{1x} \\ global_y &= I_{1y} \\ globalAngle &= \text{resizeAngle}(\Theta_{1A} + (angleDiff_{\Theta_{0A},\Theta_{0B}}/2)) \end{aligned}$$

$\diamond$

APPENDIX **E**

Velocity estimation in Soccer Server

The estimation of the velocities of moving objects observed by a given player is critical for reactive reasoning in the Soccer Server environment. For example, estimating how long it may take an agent to intercept an incoming ball depends on knowing the velocity of the ball.

For the sake of clarity, this appendix provides the derivation of the formulas used to estimate the x- and y-components of the velocity of any given moving object in the Soccer Server environment. This derivation is based on the formulas used by the Soccer Server to provide related information to client programs (i.e., players). As part of its visual feedback, the Soccer Server provides information about the distance of a moving object (such as the ball or another player) from the observer, and the change in the distance ($distChng$ below) and the change in the direction of that object ($dirChng$ below). The Soccer Server computes these values for each seen moving object, m, using the following formulas (See [Chen et al., 2002], page 28):

$$distChng = (v_{r_x} * e_{r_x}) + (v_{r_y} * e_{r_y}) \tag{E.1}$$

$$dirChng = \frac{-(v_{r_x} * e_{r_y}) + (v_{r_y} * e_{r_x})}{distance} * (180/\pi) \tag{E.2}$$

where $distChng$ is the change in the direction of object m and $dirChng$ is the change in the direction of m. (v_{r_x}, v_{r_y}) is the true current velocity of m, $distance$ is the current distance of m from the observing player, and (e_{r_x}, e_{r_y}) is a unit vector in the direction that the seen object, m, from the perspective of the observer. The unit vector (e_{r_x}, e_{r_y}) can be computed as follows:

$$unitVectorAngle = -normalizeAngle(observerGlobalAngle + (-direction)) \tag{E.3}$$
$$(e_{r_x}, e_{r_y}) = polarToUnitVector(distance, unitVectorAngle) \tag{E.4}$$

where $observerGlobalAngle$ is the current global angle of the *neck* of the observer and $direction$ is the relative offset angle of m from the neck of the observer as reported by the Soccer Server. $normalizeAngle$ is a function that normalizes an input angle value to the range $[-180 \ldots + 180]$. $unitVectorAngle$ is the angle, in Cartesian convention, of the unit vector that points in the direction of the seen object, m, from the center point of the observing player. Since the angle convention of the Soccer Server is the reverse of the counterclockwise-positive Cartesian system, we invert the $direction$ value in *Equation E.3* so that we can add it to the $observerGlobalAngle$ value, which is in Cartesian convention. Also note that we negate the return value of the $normalizeAngle$ function in *Equation E.3*. This negation is to convert the resultant angle value from the Cartesian to Server convention such that $unitVectorAngle$ is in the angle convention of the Soccer Server. $polarToUnitVector$ is a function that converts the polar coordinate description that represents the direction of a non-unit vector to the unit vector (e_{r_x}, e_{r_y}). Hence, the only unknowns in the Soccer Server formulas (*Equation E.1*, *Equation E.2*) are v_{r_x} and v_{r_x}.

Since both the $distChng$ and $dirChng$ values provided by the Server are not exact but noisy, the formulas we derive here can at best provide an estimation of the true velocity of m. In the remainder of this appendix, we derive v_{r_x} followed by v_{r_y}.

Rearranging *Equation E.2* to isolate v_{r_x} on the left-hand-side of the equation, we get:

$$\begin{aligned}
\frac{dirChng * \pi * distance}{180} &= -(v_{r_x} * e_{r_y}) + (v_{r_y} * e_{r_x}) \\
(v_{r_x} * e_{r_y}) &= (v_{r_y} * e_{r_x}) - \frac{dirChng * \pi * distance}{180} \\
v_{r_x} &= v_{r_y} * \frac{e_{r_x}}{e_{r_y}} - \frac{dirChng * \pi * distance}{180 * e_{r_y}} \qquad \text{(E.5)}
\end{aligned}$$

Rearranging *Equation E.1* to isolate v_{r_y} on the left-hand-side of the equation, we get:

$$\begin{aligned}
(v_{r_y} * e_{r_y}) &= distChng - (v_{r_x} * e_{r_x}) \\
v_{r_y} &= -v_{r_x} * \frac{e_{r_x}}{e_{r_y}} + \frac{distChng}{e_{r_y}} \qquad \text{(E.6)}
\end{aligned}$$

Substituting for v_{r_y} from E.6 into E.5, we get:

$$\begin{aligned}
v_{r_x} &= \left(-v_{r_x} * \frac{e_{r_x}}{e_{r_y}} + \frac{distChng}{e_{r_y}}\right) * \frac{e_{r_x}}{e_{r_y}} - \frac{dirChng * \pi * distance}{180 * e_{r_y}} \\
&= -v_{r_x} * \frac{e_{r_x}^2}{e_{r_y}^2} + \frac{distChng * e_{r_x}}{e_{r_y}^2} - \frac{dirChng * \pi * distance}{180 * e_{r_y}} \\
v_{r_x}\left(1 + \frac{e_{r_x}^2}{e_{r_y}^2}\right) &= \frac{distChng * e_{r_x}}{e_{r_y}^2} - \frac{dirChng * \pi * distance}{180 * e_{r_y}} \\
v_{r_x}\left(\frac{e_{r_y}^2 + e_{r_x}^2}{e_{r_y}^2}\right) &= \cdots \\
v_{r_x}\left(\frac{1 - e_{r_x}^2 + e_{r_x}^2}{e_{r_y}^2}\right) &= \cdots \\
v_{r_x}\left(\frac{1}{e_{r_y}^2}\right) &= \frac{distChng * e_{r_x}}{e_{r_y}^2} - \frac{dirChng * \pi * distance}{180 * e_{r_y}} \qquad \text{(E.7)}
\end{aligned}$$

Multiplying both sides with $e_{r_y}^2$ in E.7, we get the equation for v_{r_x}:

$$v_{r_x} = distChng * e_{r_x} - \frac{dirChng * \pi * distance * e_{r_y}}{180} \qquad \text{(E.8)}$$

Substituting for v_{r_x} from E.8 into E.6, we get the equation for v_{r_y}:

$$\begin{aligned}
v_{r_y} &= -\left(distChng * e_{r_x} - \frac{dirChng * \pi * distance * e_{r_y}}{180}\right) * \frac{e_{r_x}}{e_{r_y}} + \frac{distChng}{e_{r_y}} \\
&= -\frac{distChng * e_{r_x}^2}{e_{r_y}} + \frac{distChng}{e_{r_y}} + \frac{dirChng * \pi * distance * e_{r_x}}{180} \\
&= \frac{distChng}{e_{r_y}} * (1 - e_{r_x}^2) + \frac{dirChng * \pi * distance * e_{r_x}}{180} \\
&= \frac{distChng}{e_{r_y}} * e_{r_y}^2 + \frac{dirChng * \pi * distance * e_{r_x}}{180}
\end{aligned}$$

$$v_{r_y} = distChng * e_{r_y} + \frac{dirChng * \pi * distance * e_{r_x}}{180} \qquad \text{(E.9)}$$

◇

■