MONAD: A Flexible Architecture for Multi-Agent Control

Thuc Vu[*], Jared Go[*], Gal Kaminka[*'**], Manuela Veloso[*], Brett Browning[*]
{tdv, jgo}@andrew.cmu.edu, galk@cs.biu.ac.il, {mmv, brettb}@cs.cmu.edu

[*]School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213, USA
Tel: +1 412 268 1474

[**]Computer Science Department
Bar Ilan University
Ramat Gan 52900, Israel
Tel: +972-3-531-7233

ABSTRACT

Research in multi-agent systems has led to the development of many multi-agent control architectures. However, we believe that there is currently no known optimal structure for multi-agent control since the effectiveness of any particular architecture varies depending on the domain of the problem. Therefore, deployment of multi-agent teams would be significantly sped up by a development and deployment environment which would allow designers to easily modify the architecture. In this paper, we present a flexible team-oriented programming and execution architecture, MONAD, which integrates hierarchical behavior-based control, multi-agent coordination mechanisms, and agent-task allocation services. MONAD uses a novel scripting language that allows designers to easily modify the team structure, behavior hierarchy, applicability conditions, and arbitration methods, in pursuit of the best solution for a particular problem. We have evaluated the MONAD architecture within a well-accepted adversarial game environment, GameBots, to enable qualitative comparison of different control techniques. In this environment, we were able to rapidly design and test several teams of agents who used role, preference, or a combination of role and preference arbitration and observed that these different teams varied in their performance characteristics.

Keywords

Arbitration, Behavior-based control, Robot Teams, Teamwork, Collaboration, Team-oriented programming

1. INTRODUCTION

There is growing recognition, both in theory and in practice, that multi-agent teams can significantly benefit from the principled application of agent control architecture supporting explicit team control mechanisms. Such mechanisms can automate communication content and timing decisions, negotiations over joint decision making, coordination of activities, social organization into roles, and re-organization upon failures. This allows the human designer to focus on specifying the goal-oriented behavior of the agents, as collaborative behavior is automated to a large degree. Indeed, a number of teamwork models have been deployed successfully in complex dynamic multi-agent applications, e.g. GRATE* (Jennings 1995), STEAM (Tambe 1997), and ALLIANCE (Parker 1998).

Unfortunately, to our best knowledge, there is currently no known architecture which is recognized as optimal for all applications. Furthermore, experimenting with different team control architectures can be expensive both in design time as well as in deployment time. Coordination or negotiation mechanisms, different behavior-based control mechanisms, different team constitution schemes, all take non-trivial design and implementation time.

We present in this paper an architecture for multi-agent control, MONAD, which integrates script- and code-based off-line team-programming and team design, with a run-time coordination engine that can execute different team-control designs automatically within a behavior-based framework. Using the MONAD architecture, a designer can easily create, modify, and experiment with a wide variety of team structures.

MONAD includes an intuitive and flexible scripting language which enables designers to easily build a team in the form of an augmented behavior hierarchy. This behavior hierarchy lends naturally to the modification various parameters of a team, such as the structure of the hierarchy itself and the negotiation protocols to be used by the agents during synchronized execution of the hierarchy. To further facilitate the process of generating teams, MONAD also contains a GUI whose ultimate aim is to provide a visual means of manipulating and modifying the parameters of a team.

In order to execute the team structures specified by MONAD scripts, the MONAD architecture includes a run-time distributed behavior-based control engine called SCORE (Synchronized CoORdination Engine) which is able to execute *any* team design specified by the MONAD scripting language. The actual execution of specific behaviors are coded independently by the designer, but the coordinated multi-agent execution of these is not. Instead, SCORE automatically synchronizes the execution of behaviors across multiple agents, drawing on a user-designed reusable library of negotiation protocols that support different team-control designs. MONAD builds on earlier work on team-oriented programming (Pynadath et al. 1999, Tambe et al. 2000), and teamwork models (e.g., STEAM (Tambe 1997)) but extends them in several ways (see Section 2 for details).

MONAD is a first step in our attempt to provide a set of design and deployment tools for agents in virtual and physical environments. We hypothesize that it is possible to parameterize

AAMAS '03, July 14-18, 2003, Melbourne, Australia.

to a significant degree many of the various control mechanisms which have been reported on in the literature. Such parameterization will significantly reduce design and deployment time, since it will allow designers to use and re-use generic control templates, which would be instantiated simply by indicating the module to use and setting the parameters of the instantiation.

MONAD contributes concrete demonstration of this approach in a set of implemented, integrated, tools and services. In addition, using MONAD, we have been able to explore team control designs which use a mixture of negotiation policies within a general teamwork engine - designs that are not possible with current models.

The next sections will present related work and background on MONAD (Section 2), the details of MONAD (including off-line and on-line components—Section 3), and the results of using MONAD in a challenging virtual environment in which teams of agents can coordinate and compete in various team games (Section 4). Section 5 concludes.

2. MOTIVATION AND BACKGROUND

Our work on MONAD is motivated by our experience in developing teams of virtual and physical robots in different domains, and for different tasks. Development of such teams is a long and arduous process, and we have found ourselves experimenting with different control mechanisms to match the task at hand with appropriate control. We therefore decided to examine the challenge of creating a set of tools that would enable us to explore many different team control structures without having to redevelop a new coordination engine each time.

We were inspired in our work mainly by previous work on Team-Oriented Programming (Pynadath et al. 1999, Tambe et al. 2000). In this work, a graphical user interface is used by the team designer to construct an organizational hierarchy, assign specific tasks for organizational roles, and then assign agents to the organization. During execution, the agents utilize the STEAM teamwork model (Tambe 1997) to automatically coordinate the execution of their tasks. Unfortunately, the tools and run-time model described do not support several key components of team control, which we felt were critical.

For instance, Tambe et al.'s team-oriented programming did not allow the designer to specify the negotiation schemes that should be used by the agents at different points (Pynadath et al. 1999). Instead, a fixed protocol, built into STEAM, was used for all joint decisions. However, we believe that different situations require agents to negotiate (arbitrate) in different ways, and thus MONAD provides the designer with tools to specify different protocols and the situations in which they should be applied. This is supported at run-time by the SCORE coordination engine and modular arbitration algorithms either coded by the designer or reused from other sources. These arbitration methods can implement well-studied negotiation algorithms such as fixed-role, voting, market, preference, etc. Given an implementation of an arbitration technique, MONAD allows the designer to specify constraints and parameters for that arbitrator on a per-behavior basis. This flexibility is not possible in the earlier work.

The SCORE teamwork model and arbitration methods are also related to previous teamwork models that have been reported in the literature, such as GRATE (Jennings 1995), or ALLIANCE (Parker 1998). However, such teamwork models address only the execution-time component in deploying teams, and suffer from the same weaknesses as STEAM (Tambe 1997) with respect to being able to use different arbitration methods at different times (based on designer-specified conditions). On the other hand, models such as STEAM and ALLIANCE have mechanisms for failure detection and recovery, while MONAD and the SCORE engine currently lack support for such mechanisms.

3. MONAD ARCHITECTURE OVERVIEW

The MONAD system is composed of several key components which work together to provide a flexible framework for multi-agent programming (Figure 1). The components provided by the designer include both offline script and code, in the form of the team program, team description file, arbitration execution code, and behavior execution code. The team program and description file are both written in a format specified by the MONAD architecture, while the arbitration and behavior execution must be written and compiled to native code. However, the arbitration execution code is portable across all teams and a given arbitration algorithm, once written, can be added to user-collected libraries to be used in any team program. Given these inputs, the MONAD architecture provides synchronized execution of the team program through SCORE which acts as a distributed coordination system for the entire team. Each agent on a team runs an identical

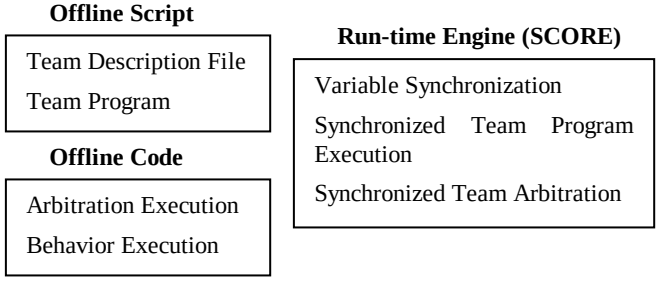

Figure 1. Components of the MONAD Architecure.

copy of SCORE and initializes it with the same team program and a team description file, along with an index which uniquely identifies to SCORE which agent it is controlling. From this point on, SCORE specifically handles the tasks of (1) ensuring that the information one agent receives about the world state is synchronized with the information known by the rest of the team (subject to the designer's wishes concerning data replication), (2) ensuring synchronous transition from one behavior to the next by means of sending the appropriate state change messages, and (3) ensuring that the correct arbitration algorithm is used and that the agents begin their execution of these algorithms in a synchronized fashion.

3.1 Formal Definitions

To aid in our explanation of the architecture, we will begin by formally defining the terminology that we use to describe various structures, relations, and algorithms as they are used in the MONAD architecture.

The team program specifies a *behavior hierarchy* that can be thought of as a Directed Acyclic Graph in which each node represents a behavior and each edge represents the relationship between two behaviors. An extracted portion of a behavior hierarchy, centered on some node A, is shown in Figure 2.

In the behavior hierarchy, there are two types of directed edges generated from one node, namely *horizontal* and *vertical* edges. A horizontal edge from one node to another represents a sequential relationship whereas a vertical edge from one node to another represents a decompositional relationship. If behavior A is connected vertically to behavior B, we say that B is a child behavior of A. Similarly, if behavior A is connected horizontally to behavior B, we say that B follows A or that B is the following behavior of A. In terms of the structure of the behavior hierarchy, the MONAD architecture forces the following restrictions: the in-degree of each node is equal to one (except for the starting behavior, which has in-degree zero), and a node can have at most one outgoing horizontal edge and any number of outgoing vertical edges. These restrictions are necessary for maintaining synchronization as will be described in detail in Section 3.3.

There are also two types of nodes in the hierarchy, leaf nodes and internal nodes. Internal nodes have one or more outgoing *vertical* edges and represent *internal behaviors. Internal behaviors* are behaviors whose goals can only be achieved through the accomplishment of a set of sub-goals which belong to their child behaviors. Conversely, leaf nodes have no outgoing vertical edges and correspond to *leaf behaviors* that specify an atomic goal and the execution required to achieve that goal.

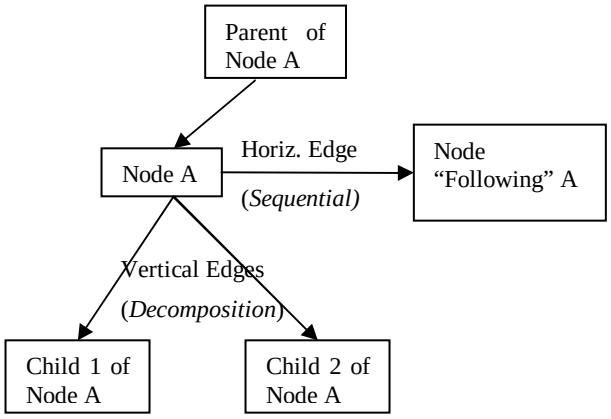

Figure 2. Relationships for Node A in a team program.

Each behavior node also has well-defined starting conditions and ending conditions that signify when it should begin and cease execution.

A *behavior stack* is a stack on which agent's currently executing behaviors are stored. The order in which items are pushed and popped from the stack is defined by the MONAD execution model (Section 3.3). Each agent maintains one behavior stack as part of its SCORE engine.

An *execution group* for a behavior is defined as a set of agents that are currently executing that behavior, or equivalently they have the behavior on their behavior stack. Every agent belongs to the execution groups of all behaviors that they have on their stack.

An *arbitration method* or *arbitrator* is defined as an algorithm that an execution group of agents can use when they must make a transition from an internal behavior to its child behaviors. In this case, they must make a joint-decision among themselves which maps the agents to child behaviors. For each agent, the arbitrator will receive the information about the environment, the agent itself, and other agents if necessary, and return the new assignment for the agent in a pre-specified fashion. During the process, it is possible for the agents to communicate between each other for additional information.

3.2 MONAD Team Programming

In order to build a team, the designer must supply the four offline components: team program, team description file, behavior execution code, and arbitration execution code. This section describes each in detail.

The team program is written in the MONAD team programming language, which was designed as a streamlined way of representing the behavior hierarchy described in the previous section. A single file specifies the entire team program which includes all behaviors in the behavior hierarchy as well as information on the pertinent team variables that SCORE must synchronize and replicate among the agents. For each behavior in the file, the designer specifies the name of the behavior, its children and following behaviors, its applicability and termination conditions, as well as its arbitrators and associated constraints. As an example, the variable replication block and a single behavior from a team program file are presented in Figure 3. It describes an "Explore" behavior with particular starting and ending conditions, as well as *following* and *children* behaviors, an arbitrator and its constraints.

```
replicate
  ourBaseKnown
  ourBasePos
  theirBaseKnown
  theirBasePos
  theirFlagState
  theirFlagPos
  theirFlagHolder
  ourFlagState
  ourFlagPos
  ourFlagHolder
end

behavior Explore
  startswhen (var ourBaseKnown == NULL || var
                                  theirBaseKnown == NULL)
  endswhen (var ourBaseKnown != NULL && var theirBaseKnown
                                  != NULL)
  following WinGame
  children ExploreOurBase ExploreTheirBase

  arbitrator role
  constraint attacker
  constraint ExploreTheirBase
  constraint defender
  constraint ExploreOurBase
end behavior
```

Figure 3. An excerpt from a MONAD team program showing the variable replication block and one behavior.

The language was designed to be a direct representation of the formal structural features of a team program as described in the previous section. The fields and keywords available within the team programming language are as follows:

startswhen *<condition>*

The *startswhen* keyword is followed by a condition that uses a Boolean expression containing variables from agent's variable table. The variables used in the expression may or may not be marked for replication. This expression is called the applicability condition. When arbitrating over the behavior, the applicability conditions are checked in order to determine whether the behavior should be considered for execution.

endswhen *<condition>*
The *endswhen* keyword is similar to the *startswhen* keyword, but specifies the conditions upon which a particular behavior should cease execution.

children *<child 1> <child 2> … <child n>*
The *children* keyword describes the list of child behaviors reachable from the current behavior. These child behaviors correspond to the vertically connected nodes reachable from the current behavior. This keyword is optional since a behavior that does not have a *children* statement is by definition a leaf node.

following *<behavior>*
The *following* keyword specifies the next same-level behavior that should be executed after the current behavior. This keyword is optional and if omitted specifies that when this behavior terminates, it should be executed again.

arbitrator *<arbitrator name>*
The *arbitrator* keyword specifies the arbitrator that should be run at the current behavior, when deciding which of its alternative child decompositions should be taken. This keyword can only be omitted if the behavior is a leaf behavior.

constraint *<string>*
The *constraints* are dependent on the arbitrator specified. There can be multiple constraint statements within the body of a single behavior, each is parsed as a string and passed as a vector of strings to the arbitrator. As an example, for fixed role arbitration, the constraint field might specify which child behavior is chosen per role. For preference-based arbitration, the constraint fields might specify how many agents are required to select each behavior (to prevent a scenario where all agents pick one of the children, yet joint execution requires some to select the other children behavior as well).

In addition to the team program, the designer must provide the team description file which determines the composition of the team. The team description file specifies the number of agents on the team, the team program to use, as well as any preset bindings that must occur (such as role bindings for each agent in the event that the designer is using role-based arbitration).

```
5
test.behavior
0 role attacker1
1 role attacker1
2 role attacker2
3 role attacker2
4 role defender
```

Figure 4. A sample MONAD team description file.

An example of a team description file for a small team using role arbitration is shown in Figure 4. In this case, the file denotes a team with 5 agents running the *test.behavior* team program, with agents 0 and 1 having their variable "role" preset to *attacker1*, agents 2 and 3 having their variable "role" preset to *attacker2*, and agent 4 having its variable "role" preset to *defender*.

To facilitate the process of generating the team description file and the team program, MONAD also includes a prototype tool, BotConfig, which provides a graphical user interface (GUI) that allow a team designer to define several aspects of the team (Figure 5). While the tool is currently biased towards role-based arbitration editing, the ultimate intent of the program is to allow visual editing of the team program. Beyond facilitating team design, a visual editor would reduce the potential for syntax and design errors in the team program and thus speed up the team design process.

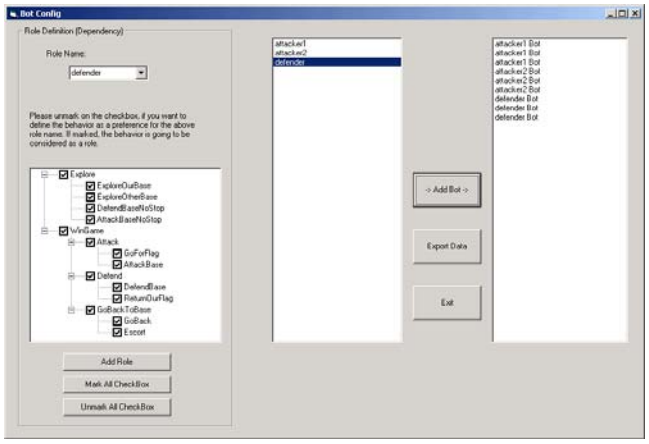

Figure 5. A screenshot of the MONAD BotConfig Tool.

Lastly, the behavior execution code and arbitration code must be written and compiled alongside the MONAD system as those systems uses constructs and data which cannot be expressed by the scripting language. For example, typical behavior execution code would involve sending particular commands to a simulation via a socket. The behavior execution and arbitration functions are registered with the SCORE engine at compile time and become available to the engine at runtime.

3.3 Execution Model and Algorithm

Given a behavior structure as previously explained, the SCORE runtime engine distributed across the team of agents provides the synchronization and execution. Within SCORE, there are two key data structures which form the cornerstone of the execution model – the behavior stack and the variable table. The behavior stack maintains the stack of behaviors as previously explained where the top behavior in the stack corresponds to the currently active behavior of the agent. The variable table contains a list of variables whose values can be modified. A subset of these variables is marked for replication by the team program. The replication itself proceeds according to a simple algorithm in each agent. When changes are made to particular variables in the table, SCORE timestamps the changes and broadcasts the new data and time of update to the other agents.

SCORE executes at fixed intervals by triggering an update cycle in which the pseudocode shown in Figure 6 is executed. At a high-level, the behavior of the agents following the update algorithm is as follows. Given a set of agents who are all executing at some node in the hierarchy (i.e. synchronized),

several important events can occur which trigger transitions in execution.

If the set of agents is executing at a leaf node, then they will simply continue execution of that node until the termination condition of one of the behaviors in their stack is met. At this point, all agents either move to the parent of the terminated behavior (if the terminated behavior has no following behavior), or move horizontally to the following behavior if it exists. In either case, once an agent transitions, it enters a waiting state in which it will not begin execution of the new behavior until all agents in the execution group of the terminated behavior have also transitioned. This ensures that all agents will begin executing the new behavior simultaneously, which is essential if the new behavior is an internal behavior and requires the execution of an arbitrator. On the other hand, if set of agents is executing at an internal node, then the agents at that node call their arbitrator methods at each iteration until each agent has enough information to determine the child behavior that it should transition to.

- The agent executes built-in code to obtain new data about its simulated surroundings and updates its internal variables accordingly.
- SCORE examines any changed variables which are marked for replication and begins broadcasting these changes to other members of the team.
- If the agent is currently ***waiting:***
  - Check if the all agents who are supposed to be in the current sub-team have joined us at the current node in the tree.
  - If all agents have joined:
    - Go to the ***executing*** state.
  - Otherwise:
    - Remain in the ***waiting*** state.
- If the agent is currently ***executing:***
  - Beginning from the bottom of the behavior stack, SCORE examines each behavior on the stack and determines if its termination condition has been met, given the value of the variables in the table.
  - If none of the conditions have broken:
    - If the current behavior is a leaf node, execute the behavior execution code for the current behavior
    - Otherwise, if the current behavior is an internal node, run the arbitration execution code specified by the current behavior. If the arbitrator completes, push the selected child behavior onto the stack and go to the ***waiting*** state. Otherwise, remain in the ***executing*** state.
  - Otherwise, on the first behavior in the stack whose termination condition was met:
    - Pop off all behaviors above and including the one which first terminated
    - If the terminated behavior has a following link, push the following behavior onto the stack
    - Go to the ***waiting*** state

Figure 6. Pseudocode for SCORE update algorithm.

Note that the set of behaviors sent into the arbitrator for consideration involve only those which are applicable, i.e. whose applicability condition *startswhen* is satisfied. Once the arbitrator returns a selection, the agent transitions into the selected behavior and remains in a waiting state until all members of the execution group have transitioned into their respective child behaviors. They then begin executing the new behavior together. In the MONAD architecture, agents are able to determine when their execution group has transitioned as the agents replicate their current behavior to the other members of the team.

As mentioned in the formal definitions (Section 3.1), the restrictions in our hierarchy are enforced in order to unify the notion of synchronized executing groups and ensure that transitions are well-defined when behaviors become no longer applicable. Many behavior-based control architectures do not differentiate between sequential and decompositional relationships as they attempt to handle sequences of actions by depending upon applicability conditions in the nodes of the hierarchy. The MONAD architecture allows for a more natural representation of temporal dependencies by separating the notion of a following relationship from a decompositional relationship. In addition, the restriction of having a horizontal out-degree of at most one allows MONAD to avoid difficulty in the following situation. If a node were followed by one or more nodes, then the relationship of those following nodes to the current node and to each other is poorly defined. The applicability and termination conditions of the two following behaviors would have no implicit relationship to the conditions of the prior behavior and thus it is possible for a subset of the following behaviors to break on some world condition when the others do not.

While this is not a direct problem, an unrestricted architecture of this form would allow multiple following behavior branches in a row. After such branching, since there is no implicit way to determine where agents should transition to when behaviors lose applicability or terminate. This would make synchronized arbitration nearly impossible as the set of agents who should be jointly deciding upon actions may fragment and be separated into different portion of the behavior hierarchy, with little unifying notion of how and when to rejoin the teams. Even in the case of adding termination links to each node of the hierarchy, there exists a large possibility of human error in creating these termination links and also a large inefficiency in explicitly specifying the transition node for each way that the behavior may have terminated.

The MONAD architecture's behavior hierarchy avoids this problem by clearly defining the two forms of relationships, decomposition and following, and using these notions to form an intuitive and implicit web of dependency between the behaviors. In our case, decomposition intuitively indicates that the goals of the child behaviors are subsets of the goals of the parent behavior. Thus, if the parent's goals are achieved and the behavior terminates, it follows that *any* behaviors in the part of the hierarchy specified by the children of the parent node should terminate since they constitute a subset of the goals of the parent node. This is achieved through our definition of execution groups and our method of examining the stack from bottom to top to determine whether or not a behavior terminated at one of our parent decomposition points. Thus, with respect to the single following relationship, we note that the restriction in the MONAD architecture forces all splits of execution groups to be decompositional. Consequently, it becomes trivial (through the behavior stack) to maintain the well-defined relationship between the agents as they split into various execution groups many levels deep, which greatly simplifies the synchronization of agents in arbitration and team program execution.

3.4 A Simple Execution Example

In order to clarify the events that occur during execution, we show the team of agents and the way in which they execute a sample team program. These agents in this example are running in a simulated GameBots Capture The Flag environment, which is explained in greater detail in Section 4. Initially, via the team description file, we set the "role" variable of each agent to be as follows: Agent 1 = "attacker1", Agent 2 = "attacker2", Agent 3 = "defender". The behavior hierarchy used in this example is shown in Figure 7. While each behavior has associated applicability, termination conditions, and constraints that are useful in our explanation, the entire team program is too long to insert in this paper and thus throughout this example we simply indicate when a particular condition breaks and the effect of the constraints on the arbitration process. For reference, Figure 3 shows the Explore behavior from this team program.

At startup, the behavior stack for each agent contains only the first behavior in the hierarchy, *Explore*, which is the initial behavior because it has in-degree 0. The strings after each agent show the

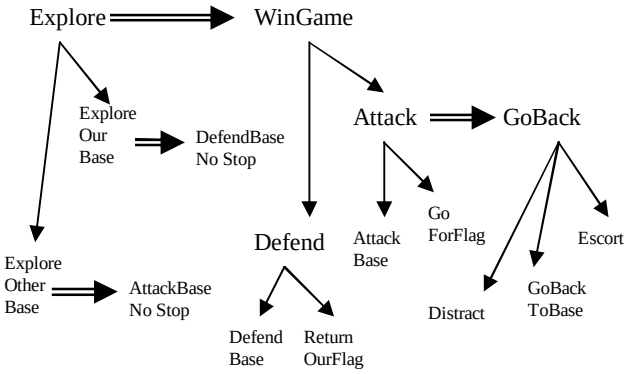

Figure 7. Behavior Hierarchy used in GameBots team and example. Double arrows indicate horizontal edges and single arrows indicate vertical edges for clarity.

behavior stack for that agent, beginning from the bottom of the stack. Therefore, the state of the team at startup is:

Agent 1 = Explore
Agent 2 = Explore
Agent 3 = Explore

At this point, the agents have just entered the game. For the purposes of this example, we will assume that the arbitration method used at all internal behaviors is a fixed-role-based arbitrator which uses predetermined assignments of agents to tasks. When the update cycle executes, since *Explore* is not a leaf node, (i.e. it has children), then there is no behavior execution code for this node. Instead, the SCORE engine in each agent evaluates the applicability condition of the two children of *Explore*, which are *ExploreOurBase* and *ExploreOtherBase*. Out of these two behaviors, the subset of behaviors whose applicability conditions evaluate to true is passed into the arbitrator so that the agents can decided amongst them. Since fixed role arbitration is being used, no communications are needed by the arbitration execution code, and the agents simply follow the constraints defined in the program. Assume for now that the constraints lead to the following assignment: Agent 1 and Agent 2 choose the *ExploreOtherBase* behavior, while Agent 3 chooses *ExploreOurBase*. The situation now looks like:

Agent 1 = Explore → ExploreOtherBase
Agent 2 = Explore → ExploreOtherBase
Agent 3 = Explore → ExploreOurBase

Now assume that the termination condition in *ExploreOurBase* becomes true when an agent on the team sees the team's flag. Then when some agent discovers the location of its team's flag, the variable *OurFlagKnown* in the variable table is set to true by the agent's update code, and is replicated to the other agents. Now, the end condition of the behavior *ExploreOurBase* becomes true, and that behavior ends. Agent 3's behavior stack is popped up to and including *ExploreOurBase*. Since the behavior *ExploreOurBase* has a following behavior (*DefendBaseNoStop*), Agent 3 pushes that behavior onto its stack, leaving the state of the team as:

Agent 1 = Explore → ExploreOtherBase
Agent 2 = Explore → ExploreOtherBase
Agent 3 = Explore → DefendBaseNoStop

This situation continues for a while, until any agent (presumably Agent 1 or Agent 2) locates the enemy team's flag. This fulfills the end condition for *Explore* since at this point the locations of both teams' flags are known. Since all three agents are in the execution group of *Explore*, they pop their behavior stacks up to and including *Explore*. Since Explore has a following behavior (WinGame), the agents push it onto their stacks:

Agent 1 = WinGame
Agent 2 = WinGame
Agent 3 = WinGame

Since WinGame has the two child behaviors *Attack* and *Defend*, it is not a leaf node and therefore the only execution for this behavior is arbitration among the three agents to determine their split into child behaviors. The agents call the role arbitrator and the constraints again cause the first two agents to transition into *Attack*, while Agent 3 transitions into *Defend*.

Agent 1 = WinGame → Attack
Agent 2 = WinGame → Attack
Agent 3 = WinGame → Defend

Since both *Attack* and *Defend* are not leaf nodes, all agents must again arbitrate. At this point, however, Agent 1 and 2 arbitrate in a separate execution group from Agent 3 since they are deciding amongst the children of *Attack* while Agent 3 decides amongst the children of *Defend*. The constraints specified at *Attack* and *Defend* then lead to following configuration:

Agent 1 = WinGame → Attack → GoForFlag
Agent 2 = WinGame → Attack → AttackBase
Agent 3 = WinGame → Defend → DefendBase

At this point, all three behaviors at the top of the stacks are leaf nodes, so at every thinking interval SCORE calls the execution function for those behaviors. Assume now at some later time that Agent 1 picks up the enemy flag. Then the end condition in behavior Attack becomes true and Agents 1 and 2 pop their

behavior stack up to and including Attack. Since Attack has *GoBack* as a following behavior, Agents 1 and 2 push *GoBack* onto their stack.

Agent 1 = WinGame → GoBack
Agent 2 = WinGame → GoBack
Agent 3 = WinGame → Defend → DefendBase

Again, the first two agents arbitrate at the *GoBack* behavior, and again the role constraints lead to Agent 1 choosing *GoBackToBase* and Agent 2 choosing *Distract*.

Agent 1 = WinGame → GoBack → GoBackToBase
Agent 2 = WinGame → GoBack → Distract
Agent 3 = WinGame → Defend → DefendBase

If the agent who is carrying the flag drops it or captures it, the end condition for *GoBack* becomes true. The first two agents would pop behaviors off their stack up to and including *GoBack*. Since *GoBack* has no following behavior, nothing is put on the stack and the Agent 1 and 2 end up returning to *WinGame* where they arbitrate.

Agent 1 = WinGame
Agent 2 = WinGame
Agent 3 = WinGame → Defend → DefendBase

Again, since we are using a fixed-role arbitrator, the arbitration execution for the first two agents leads them to both choose *Attack* and push it into their stacks. The process thus repeats until the game ends. Note that during this time, Agent 3 is executing orthogonally to the first two agents since it is in the same execution group only for *WinGame*, whose termination condition is never met until the end of the game. Furthermore, as designed, if the other team took our team's flag, then Agent 3 would have similarly transitioned to *ReturnOurFlag*, etc. without affecting the execution being carried out by Agents 1 and 2.

4. RESULTS

In order to evaluate the effects of the MONAD architecture's flexibility, we implemented a multi-agent system framework that utilizes MONAD architecture to enable the construction of different team structures which shared the low level behavior execution and arbitration execution code. We tested these different teams in a complex, dynamic environment based in the GameBots multi-agent testbed (Kaminka et al. 2002). This test-bed facilitates experiments with agents in a rich virtual environment, in which the physical characteristics of the environment and the task to be carried out by agents can be flexibly changed. For the purposes of our experiments, we chose the Capture-the-Flag task, in which two teams of agents attempt to each steal the opponent's flag from the opponent's base as many time as possible, while at the same time protecting their own flag from the opponent. The agents, each a separate program, connect to the game through sockets, using a text protocol that allows them to sense their immediate surrounding, and act (e.g., by moving, turning, etc.). The game ends when one team reaches the capture limit (2, in our case) or when the game reaches the time limit (15 minutes, in our case). When the game ends, the team that captured more flags wins. If both teams have the same number of flags, the game is considered a tie.

Using MONAD, one can make significant changes to the team structure simply by changing the configuration of edges between behaviors in the hierarchy or the associations between child behaviors and their behavior execution code. However, we focused our tests on a MONAD feature that distinguishes it from previous investigations: Its ability to allow the designer to define different multi-agent arbitration methods for use by team-members under different conditions specified on a per-behavior basis.

We created three teams from three similar team description files which all contained the general behavior hierarchy as in Figure 7 (see Section 3.2). Each behavior in the hierarchy has all the details as mentioned in Section 3.2: starts-when, ends-when conditions; the constraints of splitting and the children behaviors for the team to split into if there is any; and either the name of the arbitration method or the name of the execution function, depending on whether the behavior is at an internal node or a leaf node.

The only difference between the teams was the arbitration method that we used at the internal nodes. For the first team, ROLE, all the internal behaviors used role arbitration model. For the second team, PREF, a preference arbitration model was used. Finally, for the third team, MIXED, a combination of the previous two models was designed where one of the most important behaviors, the Attack behavior, used preference arbitration and the remaining behaviors used role arbitration.

In total, we managed to run 135 games for ROLE, 59 games for PREF, and 89 games for MIXED. In all of these games, we modified only the evaluated team: The opponent and the environment remained fixed. The results of these experiments are presented in table 1 and 2.

Table 1. The percentage of games won by each team.

	ROLE	**PREF**	**MIXED**
% of games won	58.519	35.593	56.180

Table 2. The mean, standard deviation, and normalized to mean ROLE values of different important figures in the following order: the difference in score, the time required for the teams to win a game, to capture the first flag, and to reach opponent's flag in Test Set 1.

	ROLE		**PREF**		**MIXED**	
	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std Dev.
Difference in score	0.60	1.35	-0.19	1.31	0.61	1.36
Time to win (sec)	733.7	229.3	847.0	130.6	670.8	199.7
	100%		115%		91%	
Time to capture flag (sec)	427.9	219.0	472.7	223.7	369.5	183.4
	100%		110%		86%	
Time to reach flag (sec)	130.7	60.2	145.3	78.6	113.0	53.1
	100%		111%		86%	

The overall performance of the teams in Table 1 and the average difference in score in Table 2 show a large difference between ROLE and PREF (58.5% compared to 35.5%, and -0.19 compared to 0.60). Interestingly, as the combination of features from PREF and ROLE, MIXED not only demonstrated competitive overall performance (56% of games won), but also demonstrated certain advantages when compared to ROLE. In particular, it required MIXED only 91% of the time ROLE needs to win a game, 86% of the time to capture the first flag, or 86% of the time to the opponent's flag.

While conducting the experiments, we noticed that role arbitration had a minor advantage in speed while preference arbitration had a more advantageous mapping of agents to chosen behaviors. Thus, to further evaluate the efficiency of the MIXED team, we increased the difficulty level of the opposing built-in Unreal Tournament bots while keeping all other features of the game the same. The effects of raising the difficulty were an increase the accuracy of the opponent team as well as an improvement the strategies that they used. Under this configuration, we ran 70 games for ROLE and 66 games for MIXED. The results of these experiments are presented in table 3 and 4.

Indeed, in this set of tests, the accuracy features from Preference Arbitration helped MIXED to have better overall performance than ROLE (33% of games won compared to 31% of ROLE) while still maintaining other advantages such as less time required to win a game (93.3%) or less time required to capture the first flag (81.4%). Thus, it appears that the performance of a multi-agent team in a given simulation is highly dependent on the

Table 3. The percentage of winning games and the average of the difference in score in Test Set 2

	ROLE	MIXED
% of winning	31.428	33.333

Table 4. The value, the percentage of the value compared to that of ROLE, and the standard deviation of different important figures in the following order: the average difference in score, the average time required for the teams to win a game, and to capture the first flag in Test Set 2.

	ROLE		MIXED	
	Value	Std. Dev.	Value	Std Dev.
Av. Diff. in score	-0.42	1.35	-0.22	1.58
Av. Time to win (sec)	657.8	196.9	507.5	183.8
	100%		93.3%	
Av. Time to capture flag (sec)	455.2	172.3	370.5	146.4
	100%		81.4%	

structure of the team, again despite the fact that all implementations shared the same low-level behavior execution and arbitration execution code. This demonstrates the need for testing different team structures in pursuit of one with the best performance for a given problem. The MONAD architecture can accommodate a wide variety of team structures with only small changes to the team program, which greatly accelerates and facilitates the development of multi-agent teams.

5. SUMMARY

We have presented MONAD, a set of design tools and supporting run-time architecture that allows a team-designer to quickly configure a team's control mechanism, and have this design automatically executed when the team is deployed. The design tools include a scripting language and a GUI. The run-time support includes a behavior-based architecture which integrates the SCORE teamwork model. We demonstrated the efficacy of our approach by showing how three different teams can be easily configured using our tools, and this indeed leads these teams to greatly differ in performance.

In terms of extending the MONAD architecture, one area of additional research is into synchronization of data and execution and improving the performance of the system under lossy and error-prone communication environments. With the current system, it is possible for communication difficulties to cause subsets of agents to lose synchronization and therefore cause errors or lockups in SCORE's execution of the team program. We would also like to research ways of improving SCORE's coordinated execution model to be compatible with larger teams of agents.

6. ACKNOWLEDGMENTS

This research was sponsored by Grants No. NBCHC010059 and F30602-00-2-0549. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the funding agencies.

7. REFERENCES

[1] Jennings, N. R. 1995. Controlling cooperative problem solving in industrial multi-agent systems using joint intentions. *Artificial Intelligence*, 75(2).

[2] Kaminka, G. A.; Veloso, M.; Schaffer, S.; Sollitto, C.; Adobbati, R.; Marshal, Andrew N.; Scholer, Andrew, S.; and Tejada, S. 2002. GameBots: the ever-challenging multi-agent research test-bed, In *Communications of the ACM*, January 2002.

[3] Parker, L. E., 1998. ALLIANCE: An architecture for fault-tolerant multirobot cooperation. In *IEEE Transactions on Robotics and Automation*, 14(2).

[4] Pynadath, D., Tambe, M., Chauvat, N. and Cavedon, L. 1999. Toward team-oriented programming. Proceedings of the Agents, theories, architectures and languages (ATAL'99) *workshop, published as Springer Verlag LNAI "Intelligent Agents VI"*.

[5] Tambe, M. 1997. Towards Flexible Teamwork Journal of Artificial Intelligence Research, Volume 7, Pages 83-124.

[6] Tambe, M., Pynadath, D., Chauvat, C., Das, A., and Kaminka, G. 2000. Adaptive agent architectures for heterogeneous team members Proceedings of the International Conference on Multi-agent Systems (ICMAS).