Intelligent autonomous agents need to be able to exhibit goal-directed behavior and reactivity to unexpected events in a complex, dynamic environment. Behavior based architectures decompose systems into behaviors, small independent processes each controlling some aspect of the agent’s interaction with the world. Such systems have been shown to effectively integrate goal-directed behavior and reactivity. However when a behavior based system has multiple complex goals, managing the interactions that occur between behaviors in the system and the world may become infeasible. We have implemented a multi-layered behavior based system that reduces the complexity of designing behavior based systems by allowing the designer to manipulate abstract behaviors directly. The use of abstract behaviors effectively partition the system into a number of behavior based systems, each corresponding to a specific complex goal or strategy. This report describes the architecture and, in particular, its application to Robocup, a simulated soccer competition being held at this year's International Joint Conference on Artificial Intelligence, in Japan.
7.5.1  Hierarchical Activation Trees ................................. 30
7.5.2  Hybrid Systems .................................................. 31

8  Suggested Enhancements .............................................. 32
1 Introduction

Agents are Artificial Intelligence entities that act autonomously in their environment. Much research into agents, especially recently, has focused on building agents that can act robustly in a real world environment. To act in the real world requires that an agent be able to handle complex and dynamic situations. Complex environments are difficult to handle because they are too complicated to model completely and have too many unknown factors to reliably predict the outcome of a series of actions. Dynamic environments present difficulties because they are rapidly and unpredictably changing.

To perform useful functions in such environments agents must be both pro-active and reactive. For an agent to be pro-active it must be able to choose actions directed towards achieving specific goals or specific tasks. For an agent to be reactive it must be able to respond in a timely manner to unexpected changes and unexpected scenarios in the environment.

Early traditional planning systems [29], focusing on the task of achieving goal-directed behavior, constructed complete plans (i.e. sequences of actions) for specific goals and then executed the plan. Such systems had problems with reasoning in real-time and coping with unexpected circumstances. These weaknesses led to the development of reactive planning systems which interleave planning and execution of the plans, making them more suitable to a changing environment.

In a seminal paper, Brooks [7] describes an alternative approach to handling the dynamics of the real world that involves decomposing an agent system into behaviors. Behaviors are independent processes each responsible for some specific aspect of the agent’s interactions with the world. Systems built in this manner are known as behavior based systems. Examples of behaviors in a robot that moves along a corridor may be an avoid-obstacle behavior, that is responsible for making sure the robot does not run into anything, and a move-forward behavior to keep the robot moving along the corridor. Brooks was particularly concerned with building robots, but his ideas have also been used in the design of intelligent software agents [1, 19, 2]. Brooks argued that the traditional design of intelligent systems, typically involving a symbolic model of the environment and a central reasoning component, was inappropriate for building robust agents which operated in a complex, dynamic environment. He, as well as other authors [23, 19, 18] have claimed that the behavior based architecture leads to a number of important properties such as robustness, fast reactivity and a better link between the agent and its environment.

In a behavior based system the various behaviors of an agent, each one implementing a specific capability of the agent, are connected together in some way, usually via some sort of network. Behaviors, running independently of each other, are affected by their perceptual input from their environment and may also stimulate or inhibit each other. Complex overall behavior of the agent emerges from the interactions between the behaviors and the environment. For example an avoid-obstacle behavior might inhibit a move-forward behavior when it detects an object. The consequence of this is that the robot’s forward motion is restricted when there is an obstacle in the way. At the same time the avoid-obstacle behavior is performing some action, probably turning, to get around the obstacle. The interactions between avoid-obstacle and move-forward result in the complex behavior of moving forward while avoiding obstacles.

Several authors have argued that purposeful, goal-directed behavior can be obtained from
such systems if the behaviors and the way they influence each other are carefully engineered [1, 6, 19, 22]. For example adding an obstacle-attract behavior to the robot described above can lead to, with careful engineering of the interactions, a left-wall following robot, as illustrated in Figure 1.

Figure 1: A behavior based left-wall following robot. When the sensor detects it is closer to the wall (second figure) the avoid wall behavior is activated more highly, turning the robot to the right. When the sensor detects no wall to the left the attract wall behavior is activated bringing the robot around the corner. (This example is rather contrived for purposes of illustration — the same effect can be achieved with much simpler behaviors.)

Although producing goal-directed behavior is possible in a behavior based system, this behavior can be very difficult to engineer, particularly when the system has a large number of behaviors, implementing multiple high-level goals (i.e. goals which are implemented by the interactions of multiple behaviors) [2, 19, 30]. The number of possible interactions grows exponentially with the number of behaviors in the system. Furthermore multiple goals effectively increase the number of possible influences on each behavior as a particular behavior
may be stimulated differently for different goals.

Several researchers have addressed the above problem by integrating a plan-based system above a behavior based component [11, 12, 26]. The plan-based system is used to manage the high-level goals and to co-ordinate the activation of behaviors. The behaviors in the behavior based level retain some autonomy for acting and reacting to the environment according to the current goal.

We have implemented an alternative approach, one that remains within the behavior based paradigm. The system is organised hierarchically — behaviors in upper levels of the hierarchy are used to control behaviors in the lower layers. High level goal-directed, or strategic, behavior is implemented by upper-layer behaviors, using lower-level behaviors as building blocks.

This architecture reduces the problem of managing interactions with a divide and conquer approach. The overall system can effectively be seen as being partitioned into a number of separate behavior based systems, each achieving a single goal, which can be activated or deactivated by a behavior based system in the layer above. There is a complete behavior based system active on all levels at all times. Behavior based systems on higher layers implement higher level goals. The lowest level behavior based systems interacts directly with the world, reacting to the situation according to their goal. Each behavior based system can be designed and implemented separately allowing incremental system development and testing. This architecture has allowed us to build agents which can handle multiple strategies by greatly reducing the task of behavior management.

As well as behavior management, the hierarchical organisation introduces the notion of abstract behaviors. These abstract behaviors correspond to the concept of strategies and are a convenient and powerful mechanism for building complex systems. The abstraction allows the designer to concentrate on the interactions between strategies without having to worry about the the low-level details. Selecting between appropriate strategies is performed by a complete behavior based system.

The proposed architecture has resulted in other benefits. Maes [20] has highlighted the lack of reusable components as a major problem with behavior based systems. Our architecture uses behaviors that can be used in more than one strategy: we refer to these behaviors as generic behaviors. Multiple behaviors can be created in a general way from a single generic behavior via the use of parameters. For example, a move behavior might be used twice, with different parameters (in this case different field positions) in a strategy with the most appropriate move behavior being more highly stimulated at runtime, depending on the particular situation the agent finds itself in.

This architecture has been applied to the design of a simulated soccer playing team for Robocup [17], a simulated soccer competition being held at the International Joint Conference on Artificial Intelligence, 1997. The Robocup initiative, which started as a real robot competition, is designed to be a test bed for research into agents for dynamic environments. The Robocup server sends percepts containing slightly abstracted perceptual information to client players. Players send primitive commands, such as kick or turn, to the server.

Robocup is an interesting domain for applying this architecture because a successful soccer player needs to combine high-level strategies with reactivity to a dynamic, complex world. For example, a player must be able to react, within the context of its current strategy, to the ball being unexpectedly at its feet; i.e. if the player is defending its goal, the reaction may be
to kick the ball to the wing, whereas if the player is attacking, the appropriate reaction may be to kick toward the goal.

This report describes an implementation of the multi-layered architecture for the Robocup domain. In Section 2 we describe some of the main types of agent architecture. In Section 3 we examine behavior based systems in detail. The Robocup domain is discussed in more detail in Section 4. Section 5 describes our multi-layered architecture. We present a detailed example of the system in Section 6. Section 7 evaluates the architecture, particularly focusing on our implementation for Robocup. Section 8 summarises our conclusions and suggests enhancements to our team.

1.1 An Example Scenario

To illustrate some of the issues surrounding strategic team play in the Robocup environment we introduce a scenario that we will return to throughout this report. The scenario revolves around our team having a corner kick. The team strategy divides the players into three groups: a kicker; attackers; and defenders. The kicker needs to kick the ball towards the front of the goals. Attackers, who make up the bulk of the team, need to be in offensive positions to receive the kick and attempt to score. Defenders should assume defensive positions in case the opposition gets the ball. Each player must also be able handle a situation where the ball is suddenly, unexpectedly at its feet.
A traditional approach to handling this situation may involve an explicit team strategy with each player being allocated a specific role at design time, such as striker\(^1\). Each role would have associated with it some predefined plans. One such plan may involve a sequence of steps such as move-to-front-of-goal, then watch-ball, then wait-until-the-ball-is-close, then kick. Team play occurs if all players complete their plans as required.

2 Agent Design

Agent Systems are an Artificial Intelligence paradigm for the design of complex, autonomous systems that are situated within an environment. Different authors have different ideas on exactly what constitutes an agent but most agree on a core group of concepts. Wooldridge and Jennings [31] propose the following as properties a program must have to be called an agent.

- **Autonomy**: Agent operate without the direct control of humans or others and have control over their actions.
- **Social Ability**: Agents interact with other agents.
- **Reactivity**: Agents perceive their environment and respond to changes in it in a timely fashion.
- **Pro-Active**: Agents do not simply respond to their environment: they are able to exhibit goal-directed behavior by taking the initiative.

Wooldridge and Jennings [31] group agent architectures into three broad categories. We briefly discuss each of these categories in the following sections. Further details can be found in [31, 25].

2.1 Deliberative Agents

In traditional AI approaches, an agent is made up of three parts: sensors; a central symbolic reasoning and planning system; and effectors. The central processing system uses symbolic reasoning to decide on an appropriate sequence of actions, or plan. The plan is then executed, step by step, by the effectors. Planning systems have been criticised for not scaling well when the complexity of the problem increases and for not being able to react in real time [6]. One reason for this is that the world may change while the system is doing its, often slow, reasoning. Another is that plans rely on being able to predict the outcome of a series of actions, but the world may not always behave as predicted. It may also not be possible, in some domains, to create the abstract world model upon which the system relies.

Reactive planners [14, 5] have been developed to add reactivity to traditional deliberative systems. In this type of system some plans handle system goals, while others are designed to handle specific events that require attention. Although this type of system is faster than

\(^1\)We use the term role informally, meaning a players responsibility in the side. Others have used the term in a technical sense particularly in multiagent systems[8].
traditional deliberative systems the reactivity must still come from the central reasoning system which may not be fast enough for some dynamic real world domains.

To illustrate some of the difficulties with traditional agents we return to the corner kick scenario. Generally a reactive planning agent system, based on plans, would assign agents particular roles, each role with its associated plans. The plan for an attacking player might be to move to a position just in front of the goal, turn towards the ball then watch the ball until it was near enough to run towards it and kick it. There are a number of problems that may occur here. One problem relates to robustness: if for some reason the player assigned the role of taking the kick is unable to fulfil this goal then the whole team strategy may fail. Another potential problem is if the ball is suddenly at an attacker’s feet there may be nothing in the plan of the attacker to handle this situation so the attacker would fail to take advantage of a good goal scoring opportunity. Even if the attacker had a plan to handle the contingency of the ball being unexpectedly at its feet it may not be able to react quickly enough to take advantage of the opportunity.

2.2 Reactive Systems

At the opposite end of the agent design spectrum are completely reactive systems. Reactive systems maintain no internal model of the current situation; rather, actions are chosen by referencing a lookup table of situation-action pairs based on the settings of its sensors.

---

\(^2\) Again, we are using the term informally.
Reactive systems have been found to be effective for well defined problems in unchanging domains but have been shown to be difficult to design when run-time flexibility or complex goal-directed behavior is required [23]. Ginsberg [13] argues that the number of situation-action pairs required for complex reactive systems would be prohibitive, as a new set of situation-action pairs is required for each new goal.

A reactive system approach to the corner kick scenario may involve reacting to the information that there was a corner kick by performing some predefined move. A striker’s reaction may be to run to the front of the goals. Reacting to the situation where the ball was unexpectedly at the player’s feet would be handled by another situation-reaction pair which would presumably involve kicking the ball at the goal. This reaction would be the same every time the ball was unexpectedly at the player’s feet regardless of any long term goal.

2.3 Hybrid Systems

A number of hybrid systems have been developed to overcome the perceived weaknesses of reactive and deliberative architectures [11, 12, 26]. Most hybrid architectures divide the system into layers, generally one layer for high-level planning and another for handling the details of interacting with the world. The upper layer is usually a reactive planning system and the lower layer a reactive or behavior based system. This type of architecture is designed to use each type of agent architecture at the level it seems most appropriate, i.e., deliberative planners for abstract goal-directed decision making and reactive systems for achieving a single, specific goal in the real world.

Firby [11] has proposed a hybrid system based on Reactive Action Packages (RAP’s), groups of methods for achieving a high-level goal in various situations, which a RAP Interpreter uses to control a behavior based system. A high-level deliberative system uses a symbolic representation of the world to decide on appropriate and applicable RAP’s. A RAP approach to the corner kick example might be for the high level planner to consult a symbolic model of the world and decide on an appropriate plan set, or RAP; for example it may decide that being a striker is now appropriate given its world information. The RAP interpreter sequences low-level “skill units” (similar to behaviors). The RAP actions turn skills on and off at the appropriate times so as to achieve its goal — each unit runs independently until it has fulfilled its particular task. Some low-level issues, such as reacting quickly to change, are dealt with at the level of skills. However a number of issues need to be resolved, including passing world information up to RAP and handling failure at the skill level. We briefly discuss these issues in Section 7.

3 Behavior Based Systems

A decade ago, in an attempt to develop robots that can act sensibly and robustly in the real world, Rodney Brooks developed the Behavior Based Architecture [7] for controlling robots. Brooks’ approach to the design of intelligent systems represents an important shift in paradigm from the more standard AI approaches. Brooks developed the architecture in hardware for the control of robots but others have since used similar techniques for creating software agents [19, 21, 2].
The behavior based architecture breaks an agent into behaviors. Each behavior is a small, self contained system with its own input, output and possibly internal state, for reaching and/or maintaining a particular goal [21]. The activation of a behavior varies with the settings of its inputs. The overall complex behavior of the agent is the emergent result of the interactions between behaviors and the world. There is no central control in a behavior based system.

Brooks argued against the reliance of the traditional architectures on creating an abstract representation of the world. In a behavior based system, behaviors use sensory input without attempting to abstract it or attach semantic understanding. There is no central or symbolic representation of the world in a behavior based system.

The behavior based architecture takes a radical view of how intelligence arises. Rather than thinking of intelligence as a black box, somewhere inside a system, that takes some input and returns an intelligent action, it views intelligence as an observed property of an entire system in its environment.

3.1 Properties of a Behavior Based System

Some of the important issues raised by Brooks are:

- **Emergent Intelligence**: Resulting overall behavior of a system that is not explicitly represented in a computer system or a direct result of first principles is referred to as emergent behavior [9, 24]. Behavior based systems exploit emergent behavior by allowing complex behavior to emerge from simple behaviors interacting with the world and each other [18].
• **No abstract representation of goals**: A behavior based system does not represent goals explicitly. Observed goal-directed behavior is an emergent property of the interactions between the behaviors and the world. Properties such as beliefs and goals are imposed by the observers, rather than by the presence of explicit internal representations [6].

• **Robust rather than optimal behavior**: Behavior based systems do not attempt to achieve optimal behavior; rather they aim for sensible, robust behavior under a wide range of unpredictable circumstances [6]. The reasoning behind the idea is that it is better to take ten steps to get to a door and cope with an unexpected obstacle than it is to take six steps but fail in the presence of an unexpected object. The fact that the subsumption architecture has behaviors running in parallel means that it is less likely to collapse totally given a change in the world, unlike traditional centralised systems which may fail to handle unanticipated circumstances.

• **Intelligence as an observed property**: Some researchers claim that intelligence is not an inherent property of a system but an observed property of the system’s interaction with a complex environment [2, 4, 6]. Behavior based systems use this idea by allowing each behavior to interact directly with the complexity of the world. This contrasts with the traditional approach of providing a central reasoning system with an abstracted model.

• **Inspiration from Nature**: Many of the ideas behind behavior based systems have come from other areas of science including neuroscience, ethology [30] and psychology.

### 3.2 Building behavior based agents

Building behavior based systems requires a rethink in the way we go about system development. Some important issues are:

• **Behavioral, rather than functional, decomposition**: Intelligence in a behavior based system is a result of a much different mechanism to a planning system leading to a substantially different design strategy [18]. One of the most interesting differences is that a designer does not break a high level task down into a series of steps. Instead they break it down into simple behaviors that will combine to achieve the required result. To illustrate the difference consider a corridor-following robot. In a traditional agent the sensory input would be used to create an objective, symbolic view of the world from which a planning system would find a path which a control system would follow. A behavior based approach would be to develop three behaviors: *move-forward; move-left* and *move-right*. *Move-forward* would always be active. *Move-left* would be active when sensors on the right sense something. *Move-right* would be active when sensors on the left sense something. The emergent behavior of these three interacting behaviors would be to follow a corridor.

• **Competing behaviors**: The behavior based architecture allows behaviors to compete with each other for control of the effectors rather than having a central arbiter decide which behavior should be in control. In domains where it is only sensible to have one

---

3 Ethology is the study of animal behavior.
behavior acting at once there must be some mechanism for giving control to the “most active” behavior. We refer to this mechanism as the action selection mechanism.\(^4\)

- **Using the world**: A behavior based system uses the world as its own model [6], in contrast to a planning system which attempts to represent the world symbolically. Behaviors exploit features of the environment to achieve their goal. Often a simple action that takes advantage of a complex feature of the environment appears purposeful. In a corridor following robot, discussed above, the robot does not have an internal map of the corridor, rather it directly, continually references its sensors. A change in the shape of the corridor will be reflected by a change in activation of appropriate behaviors via the sensors.

- **Parallelism**: Behaviors normally, at least conceptually, act continuously and in parallel. Many behaviors use ideas from control theory\(^5\) to continuously monitor the environment and act to maintain or achieve a goal. The inherent parallel nature of a behavior based system makes them very fast to react to change in the environment.

- **Situated reasoning**: A behavior may store values of, or some simple interpretation of, its sensors but it does not attempt to develop an objective, symbolic representation of the world. A behavior uses incoming information without attempting to “interpret” the information in the way that a planning system would. For example in an obstacle avoiding robot a high value on its left infrared sensor may make a behavior reactively fire the left-hand engine without “understanding” that there is a wall to the left, we shouldn’t get too close to walls, therefore we should move right, which requires firing the left-hand engine.

- **Incremental design**: Behavior based systems are developed from the bottom up rather than the top down. Individual behaviors are created and tested before being put together.\(^6\) The emergent behavior, resulting from the interactions between the combined behaviors, is then tested and individual behaviors refined as required. As a consequence of the way each behavior operates independently there is a complete working system, which can be tested in the real world, at all times.

### 3.3 Hierarchical Activation Trees

An extension to the behavior based model has been proposed by Blumberg, in order to address some of the problems related to behavior management [2]. Blumberg developed a hierarchical structure with general behaviors at the top and specific actions at leaf nodes. The system has been applied to the modelling of virtual creatures in a 3-D simulated world.

Blumberg’s system groups behaviors into behavior groups, organised into a loose tree-like structure. Behavior groups at the top of the structure contain more general types of behaviors whilst leaf nodes, closely linked to the environment, directly implement specific behaviors. High level behavior groups might implement things like defend while leaf nodes perform simple

---

\(^4\)See [2] for a discussion of different action selection techniques.

\(^5\)Theories behind mechanical and electrical systems for maintaining a particular state in a dynamic environment [27].

\(^6\)This bottom up approach, adding a behavior to obtain a new agent which ‘subsumes’ the previous agent in complexity, has led to the term Subsumption Architecture for Brooks’ approach [7].
tasks like *kick* or *move-to-goal*. Behaviors within a behavior group are mutually inhibiting in such a way that only one behavior will have a non-zero value after inhibition. The behavior with a non-zero value either directly acts in the environment, if it is a low level behavior or activates the behavior group which fulfills its goal. Blumberg's architecture is conceptually related to the one proposed here, however there are some important differences. Further properties of Blumberg's system are discussed in Section 7.

## 4 Robocup

Robocup [17] is a simulated soccer competition designed to be a test bed for comparing different AI techniques. A tournament is being run at this year's International Joint Conference on Artificial Intelligence, in Japan. The Robocup environment presents many of the difficulties that a real world environment presents. The competition aims to encourage research on techniques for building teams of fast agents which collaborate to solve dynamic problems.

![Figure 5: The Robocup soccer server user interface](image)

The Robocup soccer simulator is a server to which client players send commands to and receive perceptual information from. Figure 5, shows a screen shot of the Robocup interface. During each cycle each client can send up to three primitive commands, such as *kick*, which the server executes with an element of uncertainty. During each cycle the server sends a percept to each client giving the relative position and direction of the objects within the players field of view. The information in the percept is incomplete and uncertain. The server can also send percepts with messages from the referee or team mates, but this facility is restricted.

---

7 Actually Robocup started as a real robot competition.
4.1 Challenges provided by Robocup

Robocup is seen as an important test bed for AI techniques because a successful soccer team must contend with many of the difficulties that real world AI domains present, including [17]:

- **Decision making with bounded resources**: Unlike in some traditional AI problems, decision making is restricted by time. This means that usually a player cannot attempt to make the best decision but must try to find a reasonable solution quickly.

- **Incomplete Information**: Information given to players is only about a region just in front of them. This means players need to make decisions that take into account the different states the unknown part of the world may be in.

- **Adapting to changes in the world**: In the Robocup environment situations are changing rapidly so a player must be able to adapt its strategies quickly.

- **Acting on uncertain information**: Information in percepts received by the player from the server contain some uncertainty. Likewise commands sent by the client are executed with an element of uncertainty. Players must be able to deal with this uncertainty and possibly adjust their decisions accordingly.

- **Obstructive environment**: The opposition side in a game has goals incompatible with our own, rendering the environment hostile [17].

- **Real-time planning**: A sophisticated team would usually require players’ actions to be more than just a reaction to the current state of the world. This implies that players must be able to carry out a strategy or take part in a team maneuver. This strategic planning must be performed in real-time and adapt to the changes that occur in the world.

- **Cooperative Behavior**: A Robocup team must involve some sort of co-operation between players in order for team strategies to be carried out. This co-operation must be performed in real-time, with only simple, slow communication between players available. No central process controlling the team is allowed.

Although the simulator attempts to present agents with many of the difficulties that the real world presents, it is still only a simulation. An important simplification is that perception of the world is handled by the simulator with agents receiving relatively high-level abstract information.

We have decided to concentrate on the challenge of integrating high-level strategic behavior with reactivity in a complex, rapidly changing environment.

4.2 The Robocup Server

The Robocup server is a soccer simulator. The server simulates the results of all the client commands. There is a graphical interface provided for observers to watch the game. The client programs, connected to the server via sockets, control individual players. A percept is

---

8 The system can be downloaded from [http://www.csl.sony.co.jp/person/kitano/Robocup/Robocup.html](http://www.csl.sony.co.jp/person/kitano/Robocup/Robocup.html)
text sent to the client, by the server, containing a slightly abstracted version of information that the player is receiving at its sensors. The percepts either contain hear information or see information. A hear percept is made up of a sender, direction and the text of the message. A see percept is a series of object, direction, distance and relative velocity tuples.

Commands sent to the server by players are simple, atomic actions: either dash; say; turn; or kick. Commands are not guaranteed to be executed by the server, though they usually are. If more commands are sent than the competition configuration allows, extra commands are ignored without feedback to the client. Any command actually executed by the server is executed with a small, random uncertainty factor. For example a command \texttt{(turn 45)} may actually result in the player turning 41 degrees to the right. The client is not told exactly what was executed but may infer the result by analysing its next percept.

The simulator limits the rate perceptual information is sent to players and the rate at which player commands are executed. The current competition configuration specifies that a percept will be sent every 0.3 seconds and a client command executed every 0.1 seconds. The difference in percept to execution rate means that a player does not receive immediate feedback on every command.

Information sent from the server to the client provides incomplete and uncertain information about the part of the field that is in the player’s simulated field of view.\footnote{The current competition configuration has the field of view set at 90 degrees} Normally a client can ascertain its approximate position on the field and the approximate field positions of other players in its field of view by performing some simple calculations on the received information. Information received about an object deteriorates as the object’s distance from the player increases. The latest version of the server also sends clients information about objects close behind the player but our system was developed without use of this feature.

The only form of communication allowed between players is in the form of ASCII messages sent via the server. To communicate, a client sends to the server, in the same way it sends any other command, a \texttt{say} command with a message; that message is then passed asynchronously, but not immediately to all clients. Our current implementation does not use this feature.

5 A Multi-layered Behavior Based Architecture

For our Robocup players we designed a multi-layered behavior based agent architecture where each layer is a self contained behavior based system. Higher level layers implement more abstract behaviors by selecting and activating the appropriate behaviors at the next lowest level. Lower layers provide the reactivity of the system while higher layers provide the strategic reasoning. Since only the behaviors appropriate to the current strategy are active at any particular time the problems of managing interactions between behaviors is reduced.

We have implemented the architecture for a team of simulated soccer playing agents for the Robocup ’97 competition.
5.1 The Agent Design: Single Layer

The system is made up of a number of architecturally identical layers. In this section we describe a single layer of the multi-layered system.

![Layer Model](image)

Figure 6: Layer Model

5.1.1 Overview

Each layer of the system implements a complete behavior based system, consisting of: a number of generic behaviors, with their associated world information; a set of active behaviors; and three controlling processes (see Figure 6). The controlling processes are action selection, behavior instantiation and information extraction. The set of active behaviors changes dynamically over the life of the player but the generic behaviors are fixed.

In the context of our system a behavior is a small, functional control unit that controls a particular action. A group of behaviors interact with each other and with the world to achieve a strategy, or complex style of play. In our design a behavior is created dynamically, via instantiation, from a generic behavior and a parameter component. On each layer, only the command from the controlling behavior, the behavior chosen by the action selection process, is executed by the system.

Action selection cycles through the current active behavior set to determine the current “best” behavior. Each behavior consults its associated current world information to determine its own applicability. The action selection process combines the applicability factor with the persistence and priority values of each behavior in order to determine the best behavior. The priority is set by the layer above when a behavior is instantiated. The persistence value is calculated by the action selection mechanism to reduce oscillations between behaviors. The action selection process then obtains a command from the selected behavior and executes that command, either by sending a message to the Robocup server, in the case of the lowest layer, or by sending a command to the next layer down, in the case of upper layers.

Each generic behavior has associated with it some relevant information about its environment.
There is no central, complete or symbolic description of the world. The *information extraction process*, working asynchronously to the other processes, takes incoming percepts, extracts relevant information for each generic behaviour's world information (performing some simple consistency checking to ensure the reliability of the stored data), then passes the information to the behavior.

The *behavior instantiation process* receives a command from the layer above indicating a new set of behaviors, with associated priorities, that should be made active. The behavior instantiation process destroys the current active behaviors and creates a new active behavior set, whenever the upper layer active behavior changes. The active behaviors in the topmost layer are constant for the life of the agent.

The information extraction process is transparent to the action selection and behavior instantiation processes.

### 5.1.2 Generic and Parameterised Behaviors

A behavior in the architecture is made up of two parts: a *generic behavioral* part, which implements the functionality of the behavior, and a *parameter component*. We refer to the process of creating a behavior from a generic behavior and a parameter as *instantiation*. The generic part of a behavior is created once and exists for the life of the agent but parameters and the resulting instantiated behavior are created dynamically as required.

A generic behavior provides two functions — a *control* function, for issuing a command, and an *applicability* function through which the behavior indicates its relevance in light of the current state of the world. A generic behavior is designed to control a very specific action without regard to any other actions the agent may potentially execute. The narrow focus of a behavior allows the designer to write small, fast implementations.

Each generic behavior has associated with it some simple information about the world which has been extracted from the incoming percepts. Information for higher level behaviors is at a more abstract level than for lower level behaviors. For example, *defend*, an upper level behavior, may record that the ball is in the back half of the ground while *move-to-ball*, a lower level behavior, may record that the ball is 45 degrees to the left.

The use of parameters means that the same generic behavioral code can be used many times with varying effects. For example, *move* may be used with a number of different positions. One generic behavior may be part of several behaviors simultaneously, each using different parameters, as is shown in Figure 6.

A behavior issues commands and calculates its applicability without regard to the applicability or priority of other behaviors in the system. It is the designer's responsibility to ensure that the resulting interactions between behaviors are such that the required overall behavior emerges.

Implemented low level generic behaviors, for Robocup, include *kick* and *move-to*. At the next highest layer are behaviors such as *striker* and *defend*. At an even higher level of abstraction are *normal-play* and *take-kick-off*. 
5.1.3 Choosing the Best Behavior

The behavior chosen to be the controlling behavior at any particular level, i.e. the behavior whose command will be executed, is the behavior for which the sum of applicability, priority and persistence is the highest. By manipulating these three factors the designer can set up, with a relatively small number of simple behaviors, a wide range of different complex, emergent behaviors.

Applicability

A basic idea behind a behavior based system (as we described in Section 4 is that behaviors will be more or less active depending on the state of the world. In our architecture this “activation” or “excitation” is represented by the applicability value. A behavior consults its relevant world information to determine its own applicability factor. The applicability of a behavior is simply a numerical value and will vary as the world changes. The applicability of a behavior is calculated by the applicability function of a generic behavior, taking parameters into account if required, for the instantiated behavior. For example a move-to-ball behavior may have a high applicability when the ball is in view but a low applicability when the ball’s position is unknown.

Priority

Priorities are the mechanism the designer uses to control the overall behavior of the system. A priority is a property of an instantiated behavior that is constant for the life of that behavior. A player can be made to prefer attacking down the left by increasing the priority of that behavior at design time; however the player may still attack down the right if this happens to be a significantly better option, i.e. it has higher applicability, in a specific situation.

Persistence

Persistence implements the architecture’s commitment factor, by reinforcing the previously chosen behavior. Commitment is the term used to describe the tradeoff between reacting to changing situations and finishing what was started. In our system persistence is generally a comparatively small factor which only comes into play when the applicability plus priority of two or more behaviors is similar. In this case the persistence factor helps to stabilise the system and reduce oscillations between behaviors. For example when there are similar numbers of players at each end of the ground a player may oscillate between attack and defend behaviors.

Inhibition

Many behavior based systems use some form of suppression or inhibition of behaviors by other behaviors as another method for controlling interactions between behaviors. Inhibition is basically the idea of having one behavior lower the activation of another behavior. We do not use inhibition in the current implementation because we have not found it necessary. Not having inhibition also makes adding new behaviors easier because existing behaviors do not have to be changed when a new behavior is added.
5.1.4 Commands

High level behaviors issue messages to the next layer down describing the low level behaviors required to achieve the high level behaviors strategy. For example a high level attack-down-wing behavior might request low level behaviors move-to-the-ball, move-to-the-wing, pass-to-wing and kick-goal be instantiated, with corresponding priorities, in order to achieve its goal.

Behaviors at the lowest level of the system interact directly with the environment. In the Robocup domain a low-level command consists of sending an ASCII string down a socket to the Robocup server but in other domains the command may alter the power to an electric motor or draw to a screen.

5.1.5 World Information

All behaviors maintain some internal information on their local environment, but, in keeping with the behavior based paradigm, there is no global knowledge store. Since we are dealing with a simulated environment the information sent by the server, and stored by the agent, is in an abstract form. This abstraction is performed by the information extraction process. Information in higher layers is at a more abstract level than information at lower layers. For example, a low level behavior might store the $x$ and $y$ co-ordinates of the ball whereas a higher layer behavior might only store which end of the ground the ball is at.

The Robocup simulator executes client commands more often than it sends percepts\(^{10}\) meaning that often more than one command will be issued without the player receiving a percept. To allow the agent to continue to issue sensible commands the world model is updated according to the expected outcome of the command.\(^{11}\) When a percept is received the world model is reconciled with the perceived state of the world.

The uncertainty in the information, the rate at which it arrives and the fact that it only shows a small part of the field means that some reasoning must be performed to provide behaviors the best information possible. There are two types of reasoning performed by the information extraction process in an attempt to provide an accurate as possible picture of the world. The first type of reasoning removes information from the world model that cannot be reconciled with incoming information and the second type is to reason about incomplete information.

Reasoning about what cannot be true

The first way the information extraction process does reasoning is by deleting information that cannot possibly be factual. Information about an object is usually only overwritten when new information about that object is received. Having some old uncertain information about an object is typically better than having no information at all unless we know that our information is not factual, in which case it is better to have no information. For example if we last saw the ball near our goal and now, while looking at the opposition goal, we cannot see the ball it is acceptable to keep the information that the ball is near our goal because the ball is probably still there. On the other hand if we are looking at our goal, where the ball

\(^{10}\) Currently one percept to three actions.

\(^{11}\) This may not be necessary in a real world domain as the robot would have continual access to its sensors.
should be, and cannot see the ball we should update our information because the situation has obviously changed.

**Reasoning about Incomplete Information**

Handling incomplete information is more complex. For example, if our world information indicated that four of our teammates were alone at the opposite end of the ground and an incoming percept indicated that four players were at that end of the ground but the team was unknown, it would be reasonable to assume that the four unknown players were our four teammates. Our world information reasoning attempts to reconcile incomplete information with what is already in the world model. The reasoning currently performed is very simple and is an important area for further work.

### 5.2 Multi Layer Design

To provide a mechanism for engineering strategic behavior our architecture involves a number of layers, with higher layers involving greater levels of abstraction. Our layered model effectively modularises the behavior based architecture and provides a behavior management system while remaining within the basic conceptual framework. The layered model simplifies the task of managing the interactions between behaviors because a designer only ever has to handle a small number of behaviors at once. It also provides an effective mechanism for specifying high level behaviors.

The idea is that each layer is more abstract than the layer below and acts asynchronously and independently with respect to other layers. Behaviors in higher layers achieve higher level strategies by specifying and activating the lower level behaviors required to implement that strategy. All layers are architecturally identical except in the form of their input and output.

Our multi-layered architecture uses low-level behaviors to respond to unexpected changes. For example if the player’s high level goal was to move to a particular position but the ball was suddenly at the player’s feet the behavior based system would have a low-level kick-ball behavior running with a parameter indicating where to kick. The ball can be kicked without having to wait for the higher layers to change strategies.\(^\text{12}\). In contrast a deliberative system would need to have the low level information passed through the reactive layer up to the central reasoning system, a new plan be decided on then a message sent back to the reactive layer indicating what to do next. An especially nice feature of our multi-layered approach is that the reaction is influenced by the current strategy since the low level behavior that responds is partially determined by the strategy.

The layers provide the designer with an abstraction mechanism to help make the design of complex behaviors easier. In this architecture abstraction refers to the level of the behavior description. Behaviors in low levels are simple atomic actions such as kick while as we move up the layers more abstract behaviors, such as attack down the wing, are specified. This type of abstraction seems intuitive and allows a natural design process: a simple agent can be developed and higher level strategies built on top later. This type of development is especially useful in complex domains, such as Robocup, because the strategies required may not be obvious until late in development.

At first inspection it may seem as though this system differs conceptually from the standard

\(^{12}\)This is in contrast to Blumbergs [2] system — see Section 7.
behavior based ideas since only a small number of behaviors are active at once, but the layering system can be thought of as imposing a dynamic priority system on all possible behaviors. Behaviors that are not currently active have been effectively inhibited, while instantiated behaviors are at varying levels of activation.

As the number of behaviors in a behavior based system increases, prioritising behaviors becomes more complex. Our layered model alleviates this problem by having only a few behaviors on each layer active at any one time. The problem of creating a priority system among a large number of behaviors is reduced to creating several priority systems between only a few behaviors.

5.3 Final Implementation

The system was implemented in C++ under Solaris. The entire system is around 2000 lines of code, of which around 800 lines are related to processing incoming information.

Object orientated techniques have been used to allow for significant code reuse. Inheritance, abstract base classes and function overloading techniques were used to create hierarchies of generic behaviors. Each layer is an instantiation of a generic layer base class that implements all of the control systems for the layer. Action selection and behavior instantiation processes at each layer are instances of the same class. Behaviors are implemented as a pointer to a generic behavior class, a pointer to a base parameter class and a priority value.

To create a new behavior the designer simply inherits from an abstract behavior class and overrides the functions which issue a command and determine applicability. The new generic behavior has to be then declared and created in the appropriate layer. Sometimes generic behaviors inherit from other generic behaviors; for example *watch-ball* inherited from *search-for-ball* and simply provides a different applicability function.13

The current implementation has three layers: an *action* layer; a *strategy* layer; and a *style* layer. In total there are about 25 generic behaviors. At no time are more than five instantiated behaviors active at once on either the strategy or action layers. The behaviors in the bottom layer are simple actions such as *kick-to*, *move-to* and *pass-to*. These behaviors usually have a parameter providing a location for an action. The middle layer implements the strategies of a player. They include *defend*, *mid-field* and *striker*. The top layer implements the styles of play a player could be in. There are special behaviors for events like kick offs and free kicks as well as normal play. Each layer has between five and ten generic behaviors implemented for it.

13 *watch-ball* wants to always see the ball whereas *search-for-ball* wants to see it after not having seen it for a while.
6 An Example

In this section we show in detail the working of the system in the corner kick scenario we described earlier in the report.

There are seven generic behaviors required to implement the behavior required for this play. One style level behavior, *our-corner-kick*, controls the situation of taking a corner kick. It uses three strategies: *defend*, *take-corner-kick* and *striker*. The strategy level behaviors use different combinations of the following low-level behaviors to implement their strategies: *move-to-ball*, *move-to-position*, *kick* and *watch-ball*. Below we show pseudo-code for implementing each of these generic behaviors.

**Name:** our-corner-kick

**Layer:** style

**Applicability Function:**

```java
if (referee calls "our-corner-kick")
    applicability = high
else
    applicability = low
```

**Command:**

- Create: `attack`: priority = 5
- `defend`: priority = 5
- `take-corner-kick`: priority = 3

**Comment:** The *our-corner-kick* behavior creates three behaviors in the strategy level, with the given priorities. Recall these priorities remain fixed for the life of these behaviors, however, their applicability will vary.

**Name:** defend

**Layer:** strategy

**Applicability Function:**

```java
if (I am in our half)
    applicability = default-constant-value -
        (factor for every teammate near me)
else
    applicability = default-constant-value +
        (factor for every teammate near me)
```

**Command:**

- Create: `move-to-position (our-goal)`: priority = 4
- `search-for-ball`: priority = 5
- `kick (wing)`: priority = 2

**Comment:** The applicability of the defend behavior is dependent on how many team mates we have in defensive positions or attacking positions, depending on which end of the ground we are at.
**Name:** attack  
**Layer:** strategy  

**Applicability Function:**  
if (I am in their half)  
    applicability = default-constant-value -  
        (factor for every teammate near me)  
else  
    applicability = default-constant-value +  
        (factor for every teammate near me)

**Command:**  
Create : move-to-position (their-goal) : priority = 4  
    search-for-ball : priority = 5  
    kick (goal) : priority = 2

**Name:** take-corner-kick  
**Layer:** strategy  

**Applicability Function:**  
if (no one between us and the ball)  
    applicability = high  
else  
    applicability = low

**Command:**  
Create : move-to-ball : priority = 4  
    kick (front-of-goal) : priority = 2

**Name:** move-to-ball  
**Layer:** bottom  

**Applicability Function:**  
if (near, but not at, the ball)  
    applicability = high  
else  
    applicability = low

**Command:**  
if (facing the ball)  
    dash (power())  
else  
    turn (to-ball)

**Comment:** The behavior moves the player to the ball, if it is near to it. *dash* and *turn* are commands sent to the Robocup server.
Name: move-to-position (position)

Layer: bottom

Applicability Function:
    if (not in position)
        applicability = medium
    else
        applicability = low

Command:
    if (facing position)
        dash (power())
    else
        turn (power())

Name: search-for-ball

Layer: bottom

Applicability Function:
    if (can see ball)
        applicability = low
    else
        applicability = (length of time since last saw ball)

Command:
    if (can see ball)
        nothing
    else
        turn (90)

Name: kick (position)

Layer: bottom

Applicability Function:
    if (ball close enough to kick)
        applicability = high
    else
        applicability = low

Command:
    kick (direction-to (position), power-to (position))

In practice behaviors are implemented very similar to this by inheriting from and overriding parts of an abstract (in the Object Oriented sense) behavior class.

In the following pages, we step through a short period of play showing how an action is chosen on the bottom level.
There are a few things that should be noted about the above example. The low level behaviors may take more than one action to execute — e.g. move-to-position may involve a number of turn and dash commands. These actions will be interrupted if another behavior becomes more highly activated. For example, in Figure 8, move-to-position will continue to issue commands while it is the most highly activated. If the ball arrived at the player's feet while it was moving to position, and the player saw it, then the kick behavior's applicability would immediately jump and kick would start issuing commands to the server, even if move-to-position had not achieved its "goal" (i.e. the player was not in position). Also notice how sometimes it is the applicability of a behavior that determines which behavior gets control, as in Figure 7, while at other times the controlling behavior is determined by the priority, as in Figure 9.

7 Observations and Evaluation

We have implemented the architecture described in Section 5 and applied it to the Robocup domain, as described in Section 4. In this section we evaluate the architecture, particularly in how it addresses the issues raised earlier, i.e. behavior management and engineering concerns.

7.1 Behavior Management

As discussed earlier, coping with the multitude of interactions between behaviors is recognised as a serious problem for behavior based systems in complex domains or having multiple goals [19, 2]. Writing a behavior based system with a small number of behaviors and a single strategy is not particularly problematic. For example defining an appropriate activation function for a run-to-front-of-goal behavior as a part of a defend strategy is relatively straightforward because the only other active behaviors are search-for-ball and kick, which are activated by quite different situations. On the other hand, in a system requiring multiple possible concurrent strategies, such as defend and attack, defining activation functions becomes more complex. The reason for this is that the activation function of each low-level behavior must recognise not only that its specific behavior is appropriate, but also that the strategy of which it is a part is appropriate. For example, in a flat system implementing the multiple goals of attack and defend, there will be two kick behaviors active at all times, one to kick defensively and one to kick for goal. The activation function of these behaviors must not only consider whether the player is close enough to the ball to kick but also all possible factors pertaining to the choice of appropriate strategies. As each additional strategy is added, the activation function of every low-level behavior would need to be carefully engineered so it is activated at the correct time.

The layered approach we have used effectively partitions the behaviors in a layer into a number of independent behavior based systems, each implementing a single strategy. Each high-level behavior activates only enough lower level behaviors to cope with the different situations that might occur during execution of the strategy. The information about when the strategy is appropriate is encapsulated in the high-level behavior, so the applicability functions of the low-level behaviors only need to consider whether their specific behavior is appropriate; e.g. a kick behavior only needs to consider whether it is near enough to the ball to kick. Dividing the system into strategies allows the designer to consider the applicability of strategies and actions independently — activation functions for low-level behaviors are
implemented independently of the strategies that it might be a part of and the applicability functions of strategies are implemented without regard to the interactions of low-level behaviors used to implement them. As the active behaviors are only required to implement a single strategy a relatively small number is required — in practice we found that no more than five behaviors were needed to implement any given strategy in the RoboCup domain. We did not find the complexity of implementing the applicability functions for this number of behaviors problematic, although some experimentation was required to get constants right.\textsuperscript{14}

Even though behaviors are partitioned as described above, agents are able to react to unexpected changes in the world. An especially nice consequence of the multi-layered approach is that the reaction to any particular situation, by switching between lowest-layered behaviors, is partially determined by the current strategy, since this is what will have led to the bottom-layered behavior being instantiated. For example, if the ball is suddenly at the feet of a player, the \textit{kick} behavior set by the current strategy will handle the situation. A defend strategy might set a \textit{kick} behavior that kicks to the wing, whereas an attack strategy might set a \textit{kick} behavior that kicks towards goal. In this way the reaction to an unexpected circumstance is handled in a strategy-dependent way. Further, the agent does not necessarily have to change strategies to react to the change in the world.

The lack of inhibition and activation between behaviors within a layer also simplifies the behavior management because behaviors are loosely coupled from each other. Inhibition is not explicitly used, though it can be thought of as being imposed from the layer above. The only activation of a behavior comes from the internal factors of applicability, priority and persistence. It is unclear whether inhibition or activation between behaviors on a layer would help in achieving more powerful, complex behavior. All aspects of the action selection process are important areas for future work.

Change of strategy is performed at the higher, more abstract layers of the system. Behavior abstraction proved to be a very useful tool for obtaining complex behavior. The abstract behaviors used tended to correspond to conceptually intuitive strategies from the soccer domain. The use of such abstractions allows the system designer to focus on the particular conditions under which strategies should be adopted over others, without being cluttered with the implementation details. For example, the conditions when \textit{defending} is appropriate can be directly coded into the applicability function of \textit{defend} without having to worry about when exactly a \textit{defensive-kick} might be applicable.

### 7.2 Engineering Concerns

A benefit of the hierarchical architecture and the way it allows abstract representation of behaviors is that we get many of the advantages of Object Oriented design [4]. The use of generic behaviors in each layer means that significant code reuse can occur. A generic behavior is instantiated by a higher-level behavior by supplying a priority value and (possibly) parameters. This means that the same generic behavior can be used in different strategies, as well as multiple times within a single strategy (with different parameters). Because the behaviors are defined in a modular way we obtain many of the advantages of object-oriented systems, such as \textit{extendability}, \textit{maintainability} and \textit{reusable}. This is particularly relevant in the current setting where there is much experimentation and rapid prototyping required.

\textsuperscript{14} Jamie Westendorp is currently working on a GUI to help with this task.
Once a stable set of low-level behaviors had been created, a new strategy took only of the
order of ten minutes to implement. All that was required was to implement an applicability
function and specify the low-level behaviors required to implement the strategic behavior.

Once behaviors have been implemented they become building blocks for more complex
behaviors. This abstraction may be seen as providing one of the important advantages of tra-
ditional, symbolic reasoning systems, while remaining within the behavior based paradigm.
We consider the system to be behavior based rather than hybrid because behaviors, even at
higher levels, are part of a complete, independent behavior based system. A consequence of
this type of abstraction is that we were able to make abstract changes to the overall behavior
of the system very easily. For example the player may be made more attacking simply by
increasing the priority of the abstract *attack* behavior. These abstract changes could be made
easily at all levels of the system.

### 7.3 Consequences of the Behavior Based Approach

Various authors [6, 19] have claimed important properties for behavior based systems, such as
robustness, fast reactivity and appropriate behavior emerging from the behaviors’ interactions
with the environment.

We observed robustness both with respect to overcoming insufficient programmed behaviors,
as well as bugs in the system. Players in the team are usually able to respond to situations
even when behaviors have not been designed for the specific situation. For example, early on
in the development, all strategies focused on moving the ball down the centre of the ground;
however if the ball went out to the wing a player would run to the ball and the strategy
would be adapted. The overall behavior of the agent was also able to compensate for bugs in
single behaviors. Early on in development there was a bug in the function that calculated an
approximate field position from the incoming percepts. The bug meant that when the player
was outside the field looking away it would actually believe it was in the field and would often
keep running away. However the *search-for-ball* behavior would become active after the ball
was lost for a period causing the agent to look around for the ball. The field position function
would then work correctly and guide the player back into the ground, thus reducing the effect
of the bug.

Complex, seemingly planned intelligent behavior often occurred, even when it had not been
explicitly designed. For example, a player running back to a position would occasionally turn
and check where the ball was even though this behavior was not explicitly represented. This
was caused by the applicability of *search-for-ball* slowly increasing when the player could not
see the ball until it was high enough for it to overtake the *move-to-position* activation. Once
the ball was seen, the applicability of *search-for-ball* dropped again. The resulting behavior
is that a player occasionally checks where the ball is while running to a position even though
this behavior is not explicitly requested. Such behavior may seem planned to an observer.

Some emergent behavior also occurred at the team level. All team members are homoge-

---

15 Or expected!
16 This contrasts with explicit team organisation [16] where the members of the team communicate with
each other on the roles that each player should take in order to achieve the team goal. There are various
ways that distributed team organisation can be explicitly arranged but they generally require some protocol
and understanding of team tactics exist between members of the team. Simon Ch’ng[8] is using an explicit
neous. However after a settling down period different players take on roles such as striker or defender. This was due to environmental factors, e.g., a player would remain defensive because a certain number of players were already attacking, and vice versa for attackers. Though not stable this type of behavior was visible for some short periods. Different numbers of players automatically assumed each type of role depending on the total number of agents on the team. This demonstrates the robust nature of an emergent team organisation, in that it can adapt to changing scenarios.

There were some types of behavior that seemed very hard to implement in a behavior based system. An example of this is when there are two parameterised pass behaviors active at once, each wanting to pass to a different area of the field. The pass behaviors are more or less applicable depending on how sensible it is to pass to the corresponding position. The required complex behavior would be to check both positions then kick to the best position but it was not clear how to implement this in a purely behavior based method.

A sometimes observed problem with behavior based systems is the unstable oscillations between behaviors [2]. This was sometimes observed in our system. For example if there were similar numbers of players at each end of the ground there may be oscillations between defend and attack behaviors. This problem was most common when the priority values were fixed in such a way as to allow maximum flexibility in the possible behavior of a system. The oscillations were somewhat reduced by the use of the persistence factor in the behavior activation. Investigation into more effective ways to use persistence is a possible area for future work.

7.4 Performance of the Robocup Team

We tested the performance of our Robocup team by playing it against itself and against other publicly available teams. The publicly available teams ranged in standard from simple players that just followed the ball trying to kick goals, to the team that won a Robocup trial competition. We varied the number of players on a team from two through to eleven. In any distributed environment it is often hard to isolate the factors that are causing a breakdown in performance — this was the case for our Robocup team. Experimentation has led us to believe that a major factor was poor processing of incoming information; i.e., the internal state of the agent was not an accurate picture of the parts of the world the agent was interested in.

Our team was able to beat the simple teams but was beaten easily by Ogalets, the team that won the Robocup trial competition. As a soccer playing team our team was quite poor, especially the skills of the players. For example, a player attempting to defend the goal was rarely able to stop opposition attacks. Although this “defender” was not able to perform its strategy well it still behaved in a way such that an observer may view the player as having taken on the role of a defender.

An observer may also notice the players performing plan-like sequences of steps. For example when the ball came near a defender it might run to the ball, kick it away then return to its position. Although this type of behavior very rarely resulted in an effective, realistic defence of the goal there was often a clear sequence of steps, even though, again, such plans were not

\[\text{approach to team organisation for Robocup}\]

\[\text{Reynolds [28], Ferber and Drogoul [10] and others have successfully used emergent team structure to develop teams of agents for achieving tasks that appear to require global strategy.}\]
explicitly provided.

Our team was rarely able to perform successful team manoeuvres although it was often apparent to an observer what the team strategy was meant to be. For example one team strategy was to kick to the wing from defence, where a player would be waiting. This player should in turn kick the ball to the front of the goal where another player would kick the goal. Parts of this strategy were often successful but very rarely was the ball taken all the way from defence to attack without the opposition being able to intercept.

The team was comfortably beaten by a simple team, called Ogalets, whose central control was simply a set of situation-reaction pairs, implemented as an if-then-else loop. In fact, Ogalets is a particularly successful implementation of Robo cup team, comfortably winning the pre-Robo cup tournament. We believe this was due to the fact that the hard-coded behavior of this simple team did not pay much attention to its surroundings. Each player was simply allocated an area on the field, and whenever the ball came into that area it would run to the ball and kick to a hard-coded position which corresponded to a position where a team mate should be.

It is interesting to consider why a team as simple as Ogalets can perform well enough to win the pre-Robo cup tournament. The behavior of the Ogalets team is hard-coded, rigid behavior that it does not seem straightforward to extend. In the simulated environment the situation-reaction pairs work well, but real world domains are likely to require too many situation-action pairs for the building of reactive situations for strategic problems to be feasible [13]. The Ogalets team only has a single strategy; the design of appropriate situation-action pairs is analogous to the design of single strategy behavior based system, and adding more strategies is likely be quite complex, even if feasible. The Ogalets team makes no effort to be robust: if less than eleven players are on the ground there will be times when no player attempts to get the ball. The fact that Ogalets performs so well may also say something about the complexity of the simulated Robo cup environment. The environment may not actually be as significant a test bed for complex reasoning as claimed.\footnote{Or it may simply be that other implementations are only in their infancy.}

\section{Comparison to Related Work}

In this section we compare the features of our system with other related architectures. In particular we focus on architectures that use some sort of hierarchical system for managing behaviors.

\subsection{Hierarchical Activation Trees}

Blumberg [2] has developed a hierarchical behavior management system quite similar to ours which has been applied to the control of animated characters in a 3-D animated world. Blumberg’s architecture uses a tree-like structure of behaviors with more general behaviors at the top and actions at the leaf nodes. Although similar there are a number of interesting differences between the systems, including how Blumberg handles the following:

- Multiple Goals: Blumberg’s system gives increased priority to actions that leads to the fulfilment of more than one goal.
- Action Selection: Blumberg’s system uses inhibition as the only mechanism in action selection.
- Stability: Blumberg’s system uses a variable persistence factor to help stabilise the system.
- Reacting to Unexpected Circumstances: To change leaf nodes Blumberg’s system needs to traverse the entire tree.

Our action selection mechanism chooses the best behavior available and executes its command without regard to other behaviors that might also be activated. This is a property of any “winner takes all” action selection strategy. Blumberg’s architecture allows behaviors that are not selected to pass on recommendations to the layer below. This may mean an action which is appropriate to multiple medium priority goals may be selected over an action appropriate to only one slightly higher priority goal. It is not clear how this type of feature would be implemented in our architecture because of the dynamic lifetime of behaviors.

There is a significant difference in the action selection mechanisms in our system and in Blumberg’s system. In Blumberg’s system the inhibition works in such a way that only one behavior has a positive activation at any one time. This is in contrast to our system where there is no inhibition at all. We think that the mechanism of action selection is an area for future work in our system, but it is unclear whether the use of inhibition is possible, or desirable, with generic behaviors.

In our system, the persistence value provided by the action selection process is used to reduce oscillations between behaviors. In Blumberg’s system oscillations are controlled by requiring the activation of a dormant behavior to increase above the activation of the controlling behavior by more than the inhibition factor between the behaviors for a change in behavior to take place. This allows variable amounts of “commitment” depending on the particular behaviors involved, which, it seems, would have advantages over our system where the persistence factor is constant for an entire level.

For Blumberg’s system to change leaf nodes, i.e., change action, the whole tree of behaviors must be traversed. Therefore to react to an unexpected circumstance the whole tree must be considered. Our system empowers the lowest level of the system to react to changes in a strategy-dependent way. This seems to be an advantage because it is reasonable to expect that this type of reaction would be quicker than having to traverse the whole tree.

Although our architecture and Blumberg’s architecture are similar there are some significant differences. Blumberg uses a wider range of mechanisms for controlling the interactions between behaviors though it is not clear whether this leads to any significant advantages.

7.5.2 Hybrid Systems

Another popular way for achieving strategic behavior in a behavior based system is to couple a behavior based component to a deliberative system. The deliberative system activates behaviors in a similar way to higher layers in our system. Some plans, as discussed above, seem easier to code in deliberative systems while others seem easier in a behavior based style. This implies that there may be domains where hybrid systems are more suitable than pure behavior based models and vice versa. Hybrid systems may, in general, be more complex
than our system because they require two different types of architecture whereas ours uses multiple versions of the one architecture. The integration of two architectures implies that there will be more issues to resolve in order to get effective behavior. The advantage of staying within the one paradigm (in our case the behavior based paradigm) is that one sort of control and behavior module can be used at all levels: in our system the architecture at all levels is identical.\textsuperscript{19}

It is also significant that the hybrid system requires a complete, central, symbolic world model; as [7] claims obtaining this world model may not be possible in the real world. It is also claimed that the emergent behavior of a behavior based system degrades more gracefully than the behavior of a deliberative planner when unforeseen circumstances occur [6, 19]. As all levels of our system are behavior based the systems overall behavior may be more robust than that of a hybrid system (where usually only one layer is a behavior based system).

8 Suggested Enhancements

We have designed and implemented a multi-layered behavior based architecture for controlling agents in a complex dynamic domain. The architecture allows the designer to explicitly manipulate abstract behaviors, leading to improvements in the behavior management and engineering aspects of building behavior based systems. There are several areas where we believe the team could be improved or the architecture extended.

To improve the overall behavior of the Robocup team we could experiment further with the priority values and applicability functions of existing behaviors and create many new behaviors. Also, in this implementation all team members were homogeneous: the ability of the team to win soccer games might be improved by making players heterogeneous, for example having some agents with only defensive strategies and others with only attacking strategies.

There are a number of issues relating to the design of individual behaviors that warrant further investigation. At present we have only a relatively small number of strategies, each designed for a specific type of situation, so defining appropriate applicability functions has been relatively straightforward. The problem may become more complex when the number of available strategies increases. The architecture may need to be extended, possibly by introducing new layers, to handle this extra complexity.

Another possible way of obtaining more interesting overall behavior from the system may be to improve the way behaviors are activated, i.e. the action selection mechanism used. Different authors have used a wide variety of functions for combining the factors that lead to action selection. Many of these authors use some sort of inhibition between behaviors. Some work could be done on the respective advantages and disadvantages of using inhibition; i.e. the tradeoffs between lower coupling between behaviors and more control over interactions. Within our architecture investigation could be done into ways applicability, persistence and priority could be combined in a more effective way.

A real problem with the action selection has been oscillations between behaviors when information indicates that more than one behavior is applicable. Different ideas have been proposed [19, 2] but these systems rely on behaviors having knowledge of each other, which

\textsuperscript{19}All layers are actually instantiations of the same layer class.
implies a relatively close coupling between behaviors. In a more general sense work has been
done on the tradeoff between commitment and reactivity [15] but usually only in relation to
deliberative systems. In our system the problem with oscillations may be reduced by refi-
ning the persistence mechanism, perhaps by allowing the persistence value to depend on the
behavior or vary with time. (As described earlier, Blumberg’s [2] addresses this problem.)

At a more general level, it seems that some strategic behaviors are effectively described and
implemented in terms of behavior based systems while others seem to be more suited, or
only sensible, with an explicit representation. It would be interesting to see what properties
of a complex behavior made it appropriate for a behavior based system or a deliberative
system. One possible solution may be to combine a deliberative system with a multi-layered
architecture. Blumberg has combined a deliberative system with his architecture in order to
build agents for a story telling environment [3].

The hierarchical design should be applicable to other domains. In our implementation many
processes, such as action-selection, are independent of the particular domain of behaviors. It
would be interesting to look closely at the code to see whether a completely domain indepen-
dent set of classes for implementing our multi-layered architecture could be extracted.

Acknowledgements

The author would like to thank Lawrence Cavedon for all the enthusiasm and effort he has
put into this report. Thanks to Lin Padgham, Simon Ch’ng and Nick Howden for many very
interesting discussions. Also thanks to Sunny Bains, Josh Rowe, Mum and Dad for listening
patiently while I continually talked about agents. We would also like to acknowledge the
support of the Australian AI Institute for the Robocup project at RMIT.

References

cedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), Seattle,

to Animats: Proceedings of the Third International Conference on the Simulation of


Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, 1991.


Figure 7: The referee calls “our-corner-kick” (this is sent from the server as a percept) so the style level behavior \textit{our-corner-kick} gets control at the highest layer. It instantiates strategy level behaviors \textit{defend}, \textit{attack} and \textit{take-corner-kick}. Since every player hears this command, this occurs for every player. One player, noticing two players already in the attacking half, moves into a defend strategy. For the players nearest the ball the applicability of \textit{take-corner-kick} is high because they cannot see another player between them self and the ball. This behavior takes control of the middle layer, instantiating \textit{move-to-ball} and \textit{kick} at the lowest layer. The most applicable low level behavior, in this case \textit{move-to-ball}, sends an appropriate command to the server.
Figure 8: A team mate is now between the player marked with a square and the ball so the applicability of take-corner-kick decreases in the marked player. There are no visible teammates to stimulate the applicability of defend or attack. attack is selected because it has a higher priority within the take-corner-kick behavior. attack instantiates behaviors move-to-ball, kick, search-for-ball and move-to-position at the lowest level.

Figure 9: The player has moved back into a position in front of goal so the applicability of the low level behavior move-to-position has dropped. The ball is not close so neither kick or move-to-ball have high applicability. search-for-ball does not have a high applicability but its priority is high so it is selected, and the player executes actions that turn it to face the ball.
Figure 10: The ball has been kicked and is now near the player. The ball is not close enough to kick so the applicability of *kick* remains low, but the applicability of *move-to-ball* increases allowing it to select the next action. This is shown in Figure 10.

Figure 11: Finally the player finds itself near the ball. The low level behavior *kick* becomes highly activated by this situation and so takes control, kicking the ball into the goal. This kick at the goal is a reaction appropriate to the strategy.