BUILDING BLOCKS FOR AGENT DESIGN

by

Hrishikesh Jawahar Goradia

Bachelor of Engineering

University of Mumbai, 1997

Submitted in Partial Fulfillment of the

Requirements for the Degree of Master of Science in the

Department of Computer Science and Engineering

College of Engineering and Information Technology

University of South Carolina

2003

Department of Computer Science and Engineering Director of Thesis	Department of Computer Science and Engineering 2nd Reader
Department of Computer Science and Engineering 3rd Reader	Dean of the Graduate School

Abstract

Despite recent advances in the features available in current agent frameworks, designing agent-based systems remains difficult. The frameworks are implemented as large software packages, which forces software engineers to spend an unreasonable amount of time learning the package. The need for a large number of simple agents with limited abilities and short life spans that solve temporary problems in real-world applications further exacerbates the problem. We have designed and implemented an agent framework to circumvent these problems. The framework enables a user to design multiagent systems by simply wiring together the desired blocks for each agent from a pool of available components. We use the RoboCup domain as the test-bed for our research. We also define our Component-Based Agent Framework (CBAF) specifications, which attempt to merge the efforts from both agent-based and component-based software engineering methodologies. The final agent framework, SoccerBeans, is a CBAF implementation for the RoboCup domain.

Table of Contents

Abstract ii

List of Figures vi

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 3

1.3 The Solution 3

1.4 Roadmap 5

2 Background **6**

2.1 Software Engineering methodologies 6

2.1.1 Component-based Software Engineering 6

2.1.2 Agent-based Software Engineering 8

2.2 Agent Architectures 11

2.2.1 Belief-Desire-Intention Architecture 11

2.2.2 Subsumption Architecture 14

2.3 Agent Frameworks 15

2.3.1 Zeus 15

2.3.2 JADE 17

2.3.3 JAF 19

2.4 RoboCup 20

2.4.1 Soccer server 20

2.4.2 Biter 21

2.5 JavaBeans Technology 24

3 Component-Based Agent Framework 26

3.1 Domain Characteristics 27

3.2 CBAF Architecture 28

3.2.1 CBAF Specifications 29

3.2 Challenges for CBAF design 33

3.3 CBAF Implementation with JavaBeans 34

4 SoccerBeans Framework 36

4.1 Domain Analysis 36

4.2 Components for the SoccerBeans Framework 39

4.2.1 PlayerFoundation bean 39

4.2.2 Activity bean 42

4.2.3 Decision beans 44

4.2.3.1 DInputType bean 46

4.2.3.2 DClosePlayers bean 46

4.2.3.3 DClosestToBall bean 46

4.2.3.4 DBallDistance bean 47

4.2.3.5 DSelfDistance bean 47

4.2.3.6 DBallPosition bean 48

4.2.3.7 DSelfPosition bean 49

4.2.3.8 DZonalPosition bean 49

4.2.3.9 DBallUnknown bean 49

4.2.3.10 DPlayMode bean 50

4.2.3.11 DExecuteMethod bean 50

4.2.3.12 DIfThenElse bean 51

4.2.4 Behavior beans 51

4.2.4.1 BBoolean bean 52

4.2.4.2 BIncorporateObservation bean 53

4.2.4.3 BDash bean 53

4.2.4.4 BDribble bean 54

4.2.4.5 BShoot bean 54

4.2.4.6 BKick bean 55

4.2.4.7 BMove bean 55

4.2.4.8 BTurn bean 55

4.2.4.9 BCatchBall bean 55

4.2.4.10 BExecuteMethod bean 56

5 Testing and Results **57**

5.1 The sample team 58

5.1.1 Goalie ...…….…… 59

5.1.2 Defenders ...……..…….. 62

5.1.3 Defensive midfielders ...……..……... 63

5.1.4 Offensive players ...……..…….. 65

5.2 Comparisons and Results ...……..……. 69

6 Conclusions and Future Directions 70

6.1 Conclusions ……..……. 70

6.2 Future Directions ……... 72

Bibliography 74

Appendix A SoccerBeans source code 78

Appendix B SoccerBeans tutorial 79

B.1 Install the Soccer Server ……... 79

B.2 Install and Setup the BDK …… 80

B.3 Create a Soccer Player …….. 81

Appendix C Running the sample team 86

List of Figures

Figure 2.1: Schematic diagram of a generic BDI architecture ……. 12

Figure 2.2: PRS-CL Architecture ……. 13

Figure 2.3: Components of the Zeus agent building toolkit ..…….. 15

Figure 2.4: The soccer simulator display ..……... 21

Figure 2.5: Biter's architecture ……. 23

Figure 3.1. Comparison of Agent Frameworks ……. 28

Figure 3.2. The CBAF architecture ……... 33

Figure 4.1. The PlayerFoundation bean's activity scheduling algorithm ……. 40

Figure 5.1. The sample team ……. 57

Figure 5.2. Observe activity …….. 58

Figure 5.3. StartPlay activity ……. 59

Figure 5.4. GoalKick activity for goalie ……... 59

Figure 5.5. Catch activity for goalie ……. 60

Figure 5.6. Pass activity for goalie …… 61

Figure 5.7. GotoPosition activity for goalie …….. 61

Figure 5.8. Pass activity for defenders …….. 62

Figure 5.9. GotoPosition activity for defenders …… 63

Figure 5.10. Dribble activity for defensive midfielders …… 64

Figure 5.11. Pass activity for defensive midfielders ……. 64

Figure 5.12. GotoPosition activity for defensive midfielders ……... 65

Figure 5.13. Dribble activity for offensive players ……... 66

Figure 5.14. Pass activity for offensive players …… 66

Figure 5.15. GotoPosition activity for offensive center midfielder …….. 67

Figure 5.16. GotoPosition activity for other offensive midfielders …….. 68

Figure 5.17. GotoPosition activity for offensive players …….. 68

Figure 5.18. Gamecocks vs. Chikomalos …….. 69

Figure A.1. A dribbling player ……...…….. 81

Chapter 1

Introduction

1.1 Motivation

Designing multiagent systems is hard. Such systems typically contain many dynamically interacting components, each with its own thread of execution and engaged in complex coordination protocols. Efficiently engineering multiagent systems is typically orders of magnitude more complex than centralized systems that simply compute a function of some input through a single thread of control [48]. Over the years, researchers have devised concrete theories, languages and architectures for intelligent agent systems. Although many implementations that attempt to aid the development of agent-based applications exist, designing such applications from these frameworks remains a difficult task for at least two reasons. Firstly, we are limited by the design choices for the framework. While some implementations allow users to extend their frameworks for customization, such enhancements are rarely as efficient and successful as the original implementations. Secondly, the frameworks are usually huge software packages that facilitate the construction of complex, flexible and interactive agents in a distributed, concurrent environment. We need to understand the specifications for the libraries

provided for a given implementation before we can use it to design multiagent systems. As the Semantic Web and Web services become increasingly ubiquitous, we can expect systems with simple agents of limited abilities, built in order to solve temporary problems. Therefore, software engineers that have to build these agents will want methods that allow them to develop agents that have the desired capabilities quickly. They will not want to spend time learning to use complex agent architectures in order to build an agent that might be used only a dozen times. They will instead prefer to use the tools that they have grown used to, such as visual-component based development systems. Can we design an agent framework where the task of designing a multiagent system is reduced to simply wiring together the desired components for each agent without having to go through the process of learning the framework itself? Designing such an agent framework is the motivation for my thesis.

Multiagent systems are, by nature, too complex to allow a single standard design technique to work for the entire gamut of real-world application domains. Therefore, in our work, we concentrate on just a single domain: the simulation league in the RoboCup domain. This is one of the most complex multiagent domains as its run-time environment is real-time, dynamic, stochastic, partially observable, and noisy. The agents need to collaborate and work as a team to be successful. At the same time, the domain is also easy to understand, as many people are already familiar with the rules of the sport and, therefore, know what the individual agents must do in the system. These factors make the RoboCup domain an excellent test-bed for research in multiagent systems. It is fair to say

that the results derived from research on this domain can potentially be applicable to many other domains.

1.2 Problem Statement

We have attempted to develop an agent framework that can serve as a tool for creating clients for the RoboCup domain. The framework must protect the software engineers from writing the tedious, low-level programming details about the basic agent actions like dribbling and dashing, and allow them to concentrate on the research issues like coordination and planning for creating multiagent systems of soccer playing agents. In addition, the engineers must not be expected to spend time studying the underlying architecture of the framework or the provided libraries. They must be able to create the agents by simply linking the preferred components from the framework visually, in lieu of writing code. Nevertheless, they must also be able to extend the framework by developing custom components, if desired.

1.3 The Solution

We have successfully developed an agent framework for the RoboCup domain that fully satisfies all the requirements defined in the problem statement. We name our system SoccerBeans. We achieve this by merging the proven techniques of agent-based and component-based software engineering. SoccerBeans builds on our previous work, Biter [5,11,42], and inherits all the features from its predecessor. Some of the salient characteristics of the software are as follows:

1. It is based on the JavaBeans Technology, and it enables the users to develop sophisticated soccer teams *without writing any code!* Our results conclusively show that we have not traded performance to achieve this flexibility.

2. It provides many player behaviors (e.g., dashing, turning, dribbling, etc.) and potential decision criteria (e.g., ball distance, number of opponents closing in, self position on the field, etc.) as beans. Users can develop a player by simply plugging together the required components. (See the appendix for the tutorial on using SoccerBeans).

3. Users have to simply design decision tree-like plans for individual players, assuming that they play from the left side. The framework maintains an absolute-coordinate map (world model) of the soccer field and all the objects (both static and dynamic) on it. The framework also automatically adjusts the objects' coordinates for handling the change of sides. (See the appendix for the tutorial on using SoccerBeans).

4. It can be extended to incorporate other player behaviors and decisions not currently supported by the framework.

5. It has a GUI that displays the world model as pictured by a particular player at every simulation step.

1.4 Roadmap

Chapter 2 contains the background information for our work, and some of the related work in the literature. We describe our Component-Based Agent Framework (CBAF), which attempts to exploit the best features of both agent-based and component-based software engineering paradigms in Chapter 3. Chapter 4 describes SoccerBeans, which is a CBAF implementation for the RoboCup domain. We present our tests and results in Chapter 5. Finally, we analyze our work, present the conclusions, and discuss some future directions for continuing our work in Chapter 6. The appendix sections of the report provide information about retrieving the source code for the SoccerBeans package, a tutorial to jump-start the users towards using the software, and a tutorial for running the sample team provided with the package.

Chapter 2

Background

2.1 Software Engineering Methodologies

2.1.1 Component-Based Software Engineering

Generally, software reuse is one of the most effective ways of increasing productivity and improving software quality. Object-oriented methodologies promote software reuse through software architecture [37], design patterns [17], and frameworks [14], but the engineers have managed to expose the limitations for all these techniques. *Component-Based Software Engineering (CBSE)* is a software development paradigm that aims at realizing software reuse by changing both software architecture and software process [1,30]. Its goal is to compose applications by integrating existing plug & play software components on the frameworks rather than developing them. *Software components* are binary units of independent production, acquisition, and deployment that interact to form a functioning system [41].

To develop software by integrating components, components must be developed for reuse. CBSE addresses the development issues for both reusable components and

applications using the reusable components. Some of the important characteristics of component-based software development, as described in [1] are:

- *Architecture:*

Unlike conventional architectures, CBSE emphasizes modular architecture such that we can build our system incrementally by adding and/or replacing components. In order to facilitate this, the individual components are *interface-centric* (i.e. component composition is based solely on its interface and the implementation details are hidden) and *architecture-centric* (i.e. components are designed on a pre-defined architecture). Most component-based systems are built on some underlying software architecture such as the .NET framework [29], CORBA [34] and Enterprise JavaBeans [13].

- *Components:*

CBSE encompasses the entire range of components, up from generic to domain-oriented to application-specific.

- *Process:*

CBSE process consists of two tasks, component development and component integration. Software engineers would either acquire some parts of the system from the component vendors, or outsource some system parts to other organizations. Engineers can perform these tasks concurrently. The CBSE process differs from the conventional software process in that it handles a new component acquisition task.

- *Methodology:*

As discussed earlier, development methodologies of CBSE need to deal with both component development and component composition. CBSE focuses on composition of components through their interface. Composition also requires collaborative behavior of multiple components.

- *Organization:*

Since component development and component integration requires different expertise, these organizations are specialized into *component vendors* and *component integrators*. This specialization has created a new role of *component brokers*, which are mediators between the two organizations. This organization structure is called the *vendor-broker-integrator model* [30].

2.1.2 Agent-Based Software Engineering

While there is a lot of disagreement amongst researchers on the exact definition of an agent, there is a consensus on the core features that a system must exhibit to be considered as an agent. By an agent, we mean a system that enjoys the following properties – *autonomy*, *reactivity*, *pro-activeness*, and *social ability* [46,47]. We also refer to agent-based systems as multiagent systems because they usually contain numerous agents. Agent-oriented approaches can significantly enhance our ability to model, design and build complex, distributed software systems [27].

Software methodologies for the analysis and design of multiagent systems can broadly be divided into two groups [46]:

- those that take their inspiration from object-oriented development, and either extend existing object-oriented methodologies (OOM) or adapt OOM to the purposes of AOSE.
- those that adapt knowledge engineering methodologies (KEM) or other techniques.

Both the above-mentioned groups have their own strengths and weaknesses, as discussed in [23]:

- Several reasons can be cited to justify the approach of extending OOM for designing multiagent systems.
    1. The object oriented paradigm and the agent-oriented paradigm have a lot in common. [38]
    2. OOM are very popular in the industry and extensive work has already been done on, among other technologies, Object-Oriented Software Engineering [24], Object Modeling Technique [36], Object-Oriented Design [6] and Unified Modeling Language [7].
    3. The three common views of the system in OOM are also interesting for describing agents: *static* for the objects and their structural relationships, *dynamic* for describing the object interactions, and *functional* for describing the data flow of the methods of the objects. It is natural that common object-oriented languages are used to implement agent-based systems.

But on the other hand, OOM simply do not address some of the aspects of agency [38].

1. Though both objects and agents use message passing to communicate with each other, while message-passing for objects is just method invocation, agents distinguish different types of messages and model these messages frequently as speech-acts and use complex protocols to negotiate. In addition, agents analyze these messages and can decide whether to execute the requested action.
2. Agents can be characterized by their mental state, and object-oriented methodologies do not define techniques for modeling how the agents carry out their inferences, their planning process, etc.
3. Agents are also characterized by their social dimension, and procedures for modeling these social relationships between agents have to be defined.

- KEM deal with the development of knowledge based systems. Since the agents have cognitive characteristics, these methodologies provide an excellent basis for modeling this agent knowledge.

  However, there are some obvious disadvantages with this approach:

  1. Only the knowledge acquisition process is addressed in these methodologies.
  2. KEM are not as extendible as OOM.

  Despite these deficiencies, KEM have been successfully applied to many projects [23].

Over the years, researchers have managed to devise concrete models for designing various types of multiagent systems. We discuss this issue in the next section on Agent Architectures. Currently popular agent frameworks include JADE [25] and ZEUS [33]. We discuss about these frameworks in the Agent Frameworks section later in this chapter.

2.2 Agent Architectures

Agent architectures deal with the issues surrounding the construction of computer systems that satisfy the properties specified by agent theorists. We describe here two of the most popular agent architectures that are related to our work.

2.2.1 Belief-Desire-Intention architecture

The BDI architecture [19] assigns beliefs, desires and intentions to each agent. Belief represents the agent's knowledge, desire represents the agent's goals, and intention lends deliberation to the agent. In addition, the BDI architecture also defines functions that represent agent deliberation for deciding what to do and means-end reasoning for deciding how to do it. Figure 2.1 shows a schematic diagram for a generic BDI architecture.

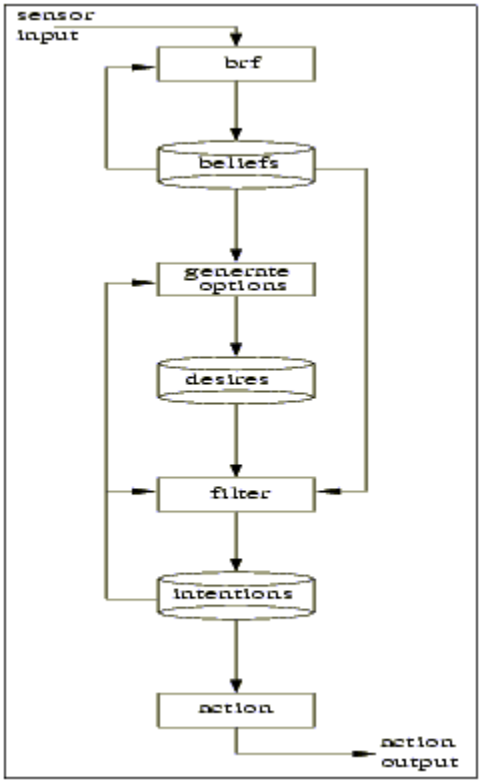


Figure 2.1: Schematic diagram of a generic BDI architecture, produced from [19]

Many practical implementations of the BDI architecture exist. Consider the Procedural Reasoning System (PRS-CL) [19,35] developed at SRI International. Figure 2.2 describes the PRS-CL architecture.

A system built in PRS-CL is intended to simultaneously achieve any goals it might have based on its current beliefs about the world while noticing and responding to new events. The architecture of PRS-CL consists of

- a database containing current beliefs or facts about the world
- a set of current goals to be realized

- a set of plans, called Acts, describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations
- an intention structure containing the plans that have been chosen for eventual execution

An interpreter manipulates these components, selecting appropriate plans based on the system's beliefs and goals, placing those selected on the intention structure, and executing them.

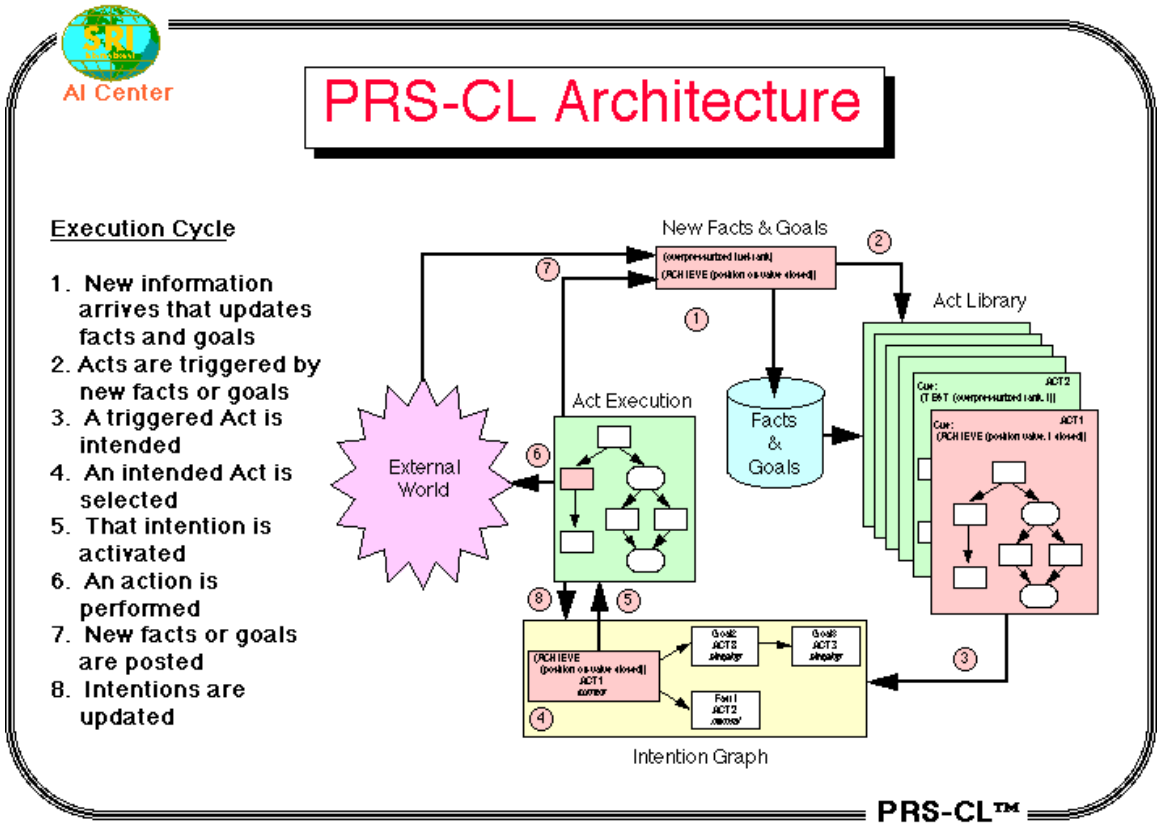


Figure 2.2: PRS-CL Architecture, produced from [35]

The PRS-CL interpreter runs the entire system. The interpreter repeatedly executes the set of activities depicted in the figure above. At any particular time, certain goals are

established and certain events occur that alter the beliefs held in the system database (1). These changes in the system's goals and beliefs trigger (invoke) various Acts (2). One or more of these applicable Acts will then be chosen and placed on the intention graph (3). Finally, PRS-CL selects a task (intention) from the root of the intention graph (4) and executes *one step* of that task (5). This will result either in the performance of a primitive action in the world (6), the establishment of a new subgoal or the conclusion of some new belief (7), or a modification to the intention graph itself (8). At this point the interpreter cycle begins again: the newly established goals and beliefs (if any) trigger new Acts, one or more of these are selected and placed on the intention graph, and again an intention is selected from that structure and partially executed.

2.2.2 Subsumption architecture

The Subsumption architecture [8] is a reactive agent architecture developed by Rodney A. Brooks, where the problem is decomposed into layers corresponding to levels of behavior. Within this setting is introduced the idea of subsumption, that is, that more complex layers could not only depend on lower, more reactive layers, but could also influence their behavior. The resulting architecture is one that could service simultaneously multiple, potentially conflicting goals in a reactive fashion, giving precedence to high priority goals.

2.3 Agent Frameworks

Agent frameworks provide a code base for developing intelligent, distributed, and autonomous software, using agents as the unit of encapsulation [12,15]. Any agent interacting within a society commonly needs a set of actions or utilities regardless of the application domain. Agent frameworks provide these utilities, thereby alleviating agent programmers from reinventing these agent behaviors. We discuss two of the most popular agent frameworks, Zeus and JADE, in this section. We also consider UMass's JAF, which is a component-based agent framework.

2.3.1 Zeus

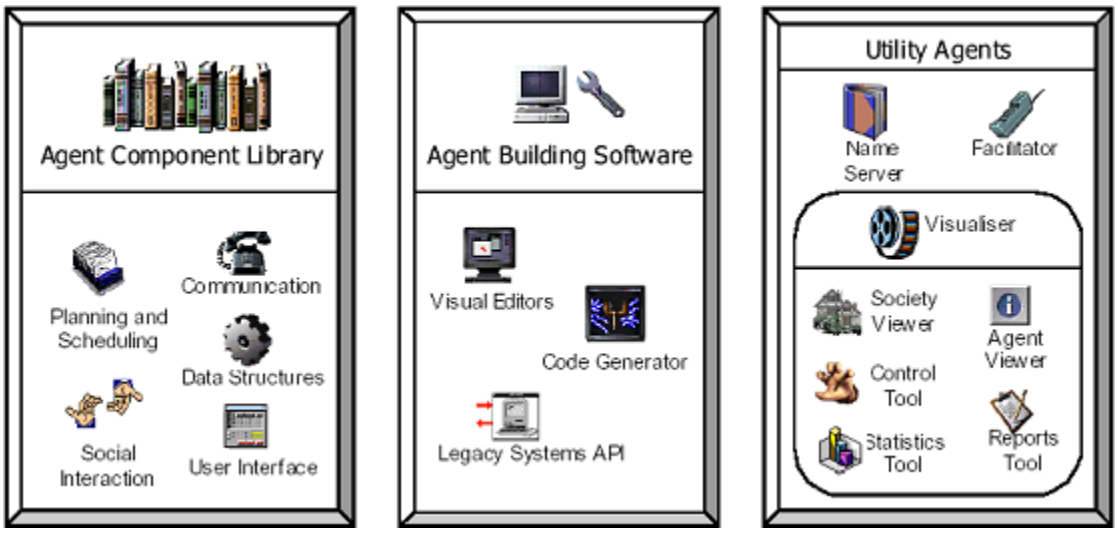


Figure 2.3: Components of the Zeus agent building toolkit, produced from [33]

The Zeus [33] Agent Building Toolkit facilitates the rapid development of collaborative agent applications through the provision of a library of agent-level components and an

environment to support the agent building process. The toolkit is a synthesis of established agent technologies with some novel solutions to provide an integrated collaborative agent building environment. Zeus defines a multiagent system design approach and supports it with a visual environment for capturing user specifications of agents that are used to generate Java source code of the agents.

The Zeus toolkit consists of a set of components, written in the Java programming language, that can be categorized into three functional groups (or libraries) as depicted in Figure 2.3: an agent component library, an agent building tool and a suite of utility agents comprising nameserver, facilitator and visualiser agents.

- *The Agent Component Library:*

The Agent Component Library is a collection of classes that form the building blocks of individual agents. Together these classes implement the application-independent *agent-level* functionality required of collaborative agents.

- *The Zeus Agent Building Approach and Environment:*

In Zeus, the agents are composed of five layers: API layer, definition layer, organizational layer, coordination layer, and communication layer. The API layer allows interaction with the non-agentized world. The definition layer manages the task the agent must perform. The organizational layer manages the knowledge concerning other agents. The coordination layer manages coordination and negotiation with other agents. Finally, the communication layer allows the communication with other agents. The Zeus

environment supports a suite of integrated editors that support the agent design approach. To facilitate ease of use, the editors are designed to enable users to interactively create components by visually specifying their attributes.

- *The Zeus Utility Agents:*

The Zeus suite of utility agents provides the infrastructure of a multiagent system. It consists of a *nameserver* and a *facilitator* agent for information discovery and a *visualizer* agent for visualizing or debugging societies of Zeus agents.

2.3.2 JADE

JADE (Java Agent Development Environment) [2,3,25] is a software framework to build agent systems for the management of networked information sources in compliance with the FIPA [16] specifications for interoperable intelligent multiagent systems. JADE simplifies development while ensuring standard compliance through a comprehensive set of system services and agents. JADE is thus an agent middleware that implements an efficient agent platform and a development framework. It deals with all the aspects that are peculiar to the agent internals and that are independent of the applications, such as message transport, encoding and parsing, or agent life-cycle management.

JADE offers the following list of features to the agent programmer:

- FIPA-compliant Agent Platform, which includes the AMS (Agent Management System), theDF (Directory Facilitator), and the ACC (Agent Communication

Channel). All these threeagents are automatically activated at the agent platform start-up;

- Distributed agent platform. The agent platform can be split on several hosts (provided that there is no firewall between them). Only one Java application, and therefore only one Java Virtual Machine, is executed on each host. Agents are implemented as one Java thread and Java events are used for effective and light-weight communication between agents on the same host. Parallel tasks can be still executed by one agent, and JADE schedules these tasks in a more efficient (and even simpler for the skilled programmer) way than the Java Virtual Machine does for threads;
- A number of FIPA-compliant DFs (Directory Facilitator) can be started at run time in order to implement multi-domain applications, where the notion of domain is a logical one as described in FIPA97 Part 1;
- Programming interface to simplify registration of agent services with one, or more, domains (i.e. DF);
- Transport mechanism and interface to send/receive messages to/from other agents;
- FIPA97-compliant IIOP protocol to connect different agent platforms;
- Light-weight transport of ACL messages inside the same agent platform, as messages are transferred encoded as Java objects, rather than strings, in order to avoid marshalling and unmarshalling procedures. When sender or receiver do not belong to the same platform, the message is automatically converted to /from the FIPA compliant string format. In this way, this conversion is hidden to the agent implementers that only need to deal with the same class of Java object;
- Library of FIPA interaction protocols ready to be used;

- Automatic registration of agents with the AMS;
- FIPA-compliant naming service: at start-up agents obtain their GUID (Globally Unique Identifier) from the platform;
- Graphical user interface to manage several agents and agent platforms from the same agent. The activity of each platform can be monitored and logged.

2.3.3 JAF

JAF [22] is a component-based architecture developed at the University of Massachusetts, Amherst, for designing the agents used within multi-agent systems. JAF facilitates code reuse and simplifies agent construction by building up a pool of components that can be easily combined in different ways to produce agents with different capabilities. JAF builds upon Sun's Java Beans component model by adding additional implementation and runtime support designed to produce consistent and cohesive components. Agents using JAF are composed of a number of such components, which interact with one another using both passive event stream monitoring and active method invocation.

The generic agent consists of 6 components: Communicate, Control, Execute, Log, ProgramSolver, and State. The ProblemSolver component encapsulates the behavior of the agent, making use of Communicate and Execute to perform messaging and action tasks, respectively. Control directs the initialization and activity of each component, and State provides access to required component references and data. All components use Log, which serves as a central location for free-form text and event logging. The Multi

Agent Survivability Simulator, a flexible execution environment that simulates the faulty or hostile conditions, is the test bed for JAF agents.

2.4 RoboCup

The RoboCup domain has presented a challenge for researchers from not only the traditionally artificial intelligence related fields but also many areas of robotics, sociology, real-time mission critical systems, etc. [31,39]. As mentioned earlier in the report, it is an excellent test-bed for the study and research of multiagent systems.

2.4.1 Soccer server

The RoboCup soccer server was developed by Itsuki Noda [32] and has been used as the basis for many successful international competitions and research challenges. Unlike many AI domains, the soccer server is a complex and realistic domain that incorporates as many real-world complexities as possible. It models a hypothetical robotic system, merging characteristics from different existing and planned systems as well as from human soccer players [40]. The server's sensor and actuator noise models are motivated by typical robotic systems, while many other characteristics, such as limited stamina and vision, are motivated by human parameters.

The soccer simulator provides an environment for agents (also referred to as clients or players) to execute. Client programs connect to the server via UDP sockets, each controlling a single player. The soccer server simulates the movements of all of the

objects in the world, while each client acts as the brain of one player, sending movement commands to the server. The server causes the player being controlled by the client to execute the movement commands and sends sensory information from that player's perspective back to the client. The simulator provides a visualization tool, the monitor, pictured in Figure 2.4.

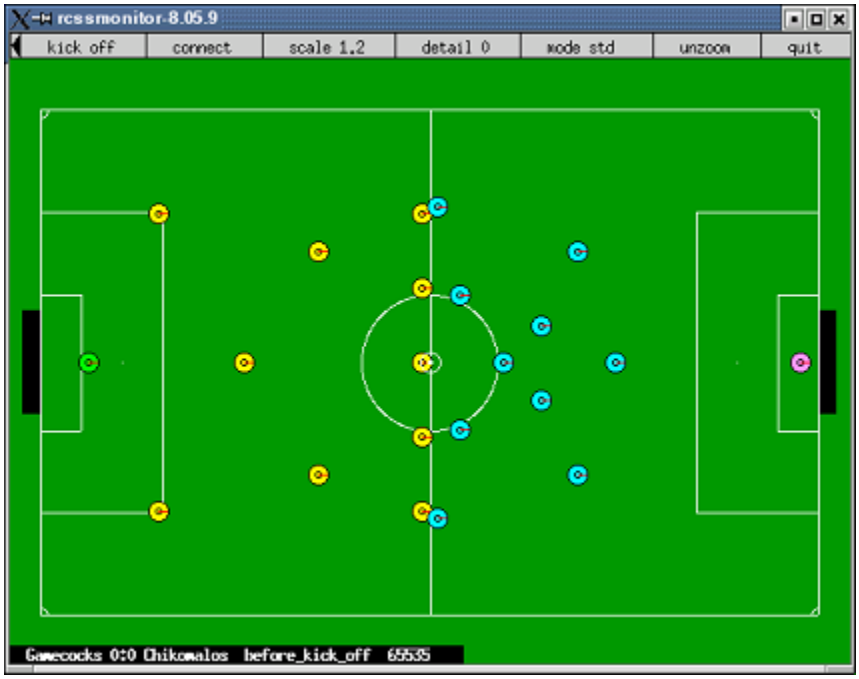


Figure 2.4: The soccer simulator display

2.4.2 Biter

Biter [5,11,42] is a basic RoboCup soccer client developed by Paul Buhler, Shaun Wood and Jose Vidal. It has been used successfully for 4 semesters in a graduate level course on multiagent systems at our university. It is a very flexible and powerful tool for agent-oriented software engineering in the RoboCup domain. The software is based on the Generic Agent Architecture [43], which is capable of handling many different activities

like reactive responses, long-term behaviors, and conversations with other agents. Biter also provides the users with an absolute-coordinate world model, a set of low-level ball handling skills, a set of higher-level skill-based behaviors, and many utility methods which allow the users to focus more directly on planning activities.

The *Generic Agent Architecture (GAA)* is an elegant object-oriented design meant to handle the type of activities typical for an agent in a multiagent system. The architecture incorporates the functionality that enables users to design either a completely reactive system or a completely planned system or a system with both reactive and planned long-term behaviors.

The GAA provides a mechanism for scheduling activities each time the agent receives some form of input. The architecture supports two types of activities, *behaviors*, which are actions taken over a series of time steps, and *conversations*, which are series of messages exchanged between agents. Biter also provides many higher-level behaviors by extending the *Behavior* class. The *ActivityManager* handles the scheduling of activities based on the state of the world and the agent's internal state. The *Activity* abstract class has three main member functions: *handle, canHandle* and *inhibits*. By manipulating these functions, the user can have both reactive and planned actions in the system. Figure 2.5 shows a general overview of the system.

The GAA can receive three types of inputs, *sensory inputs* which are the inputs sent by the server to all the agents, *messages* which are the inputs from other agents, and *events*

which are the alarms generated by the agent itself. Biter implements a special *act* event, which keeps each agent's communication, synchronized with the soccer server.

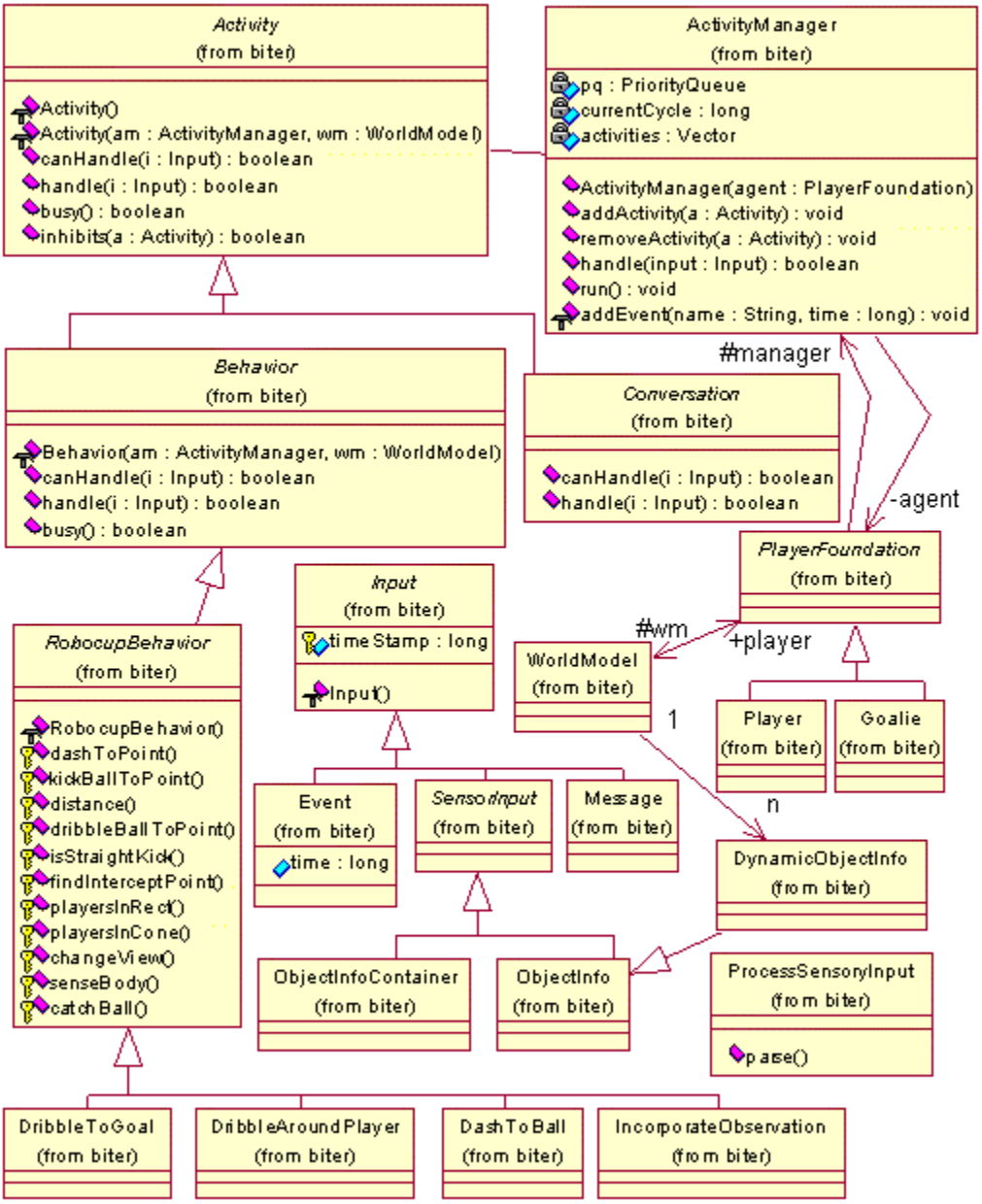


Figure 2.5: Biter's architecture, produced from [43]

Biter provides a very versatile W*orld Model* that includes an absolute-coordinate representation of both kind of objects: *static objects*, such as flags, lines and goals, and *dynamic objects*, such as the ball and players.

A player learns about its environment through the sensory input from the server. The input is comprised of vectors that point to the static and dynamic objects in its field of view. The information is processed and transformed into absolute positions for the objects in the field. Additional functionality like automatic updating for lost communication over the network, maintenance and usage of a history of recent environmental changes, etc. is provided to increase the accuracy of the model.

Troubleshooting multiagent systems can be intimidating. Biter provides a very powerful debugging feature in the graphical display of a player's internal view of its environment. The visual is a scaled-down version of the static and dynamic objects in the player's range at each time step.

Biter's world model also provides a host of other utilities like socket communication with the soccer server, incorporation of server and client configuration changes, efficient access to agent's data using regular expressions, etc.

2.5 JavaBeans Technology

The Sun JavaBeans specification [26] defines a bean as *a reusable software component that can be manipulated visually in a builder tool*. Software components are self-

contained, reusable software units. The builder tools include web page builders, visual application builders, GUI layout builders, server application builders, or simply document editors that include beans as part of a compound document. Using the builder tools, the software components can be composed into applets, applications, servlets, and composite components. Thus, JavaBeans can be simple GUI elements such as buttons and sliders, or sophisticated visual software components such as database viewers or data feeds. Some beans may have no GUI appearance of their own, but may still be composed together visually using an application builder.

Chapter 3

Component Based Agent Framework

Despite the recent advances in the quality and the available features in current frameworks, designing agent-based systems remains difficult. Both the rigid design specifications of the framework and the necessity to understand the framework's implementation details have restricted the usability of current frameworks for designing multiagent systems. The Component-Based Agent Framework (CBAF) specifications enable a software engineer to design individual agents by just associating it visually to the desired components and, thereby, circumvent the above-mentioned problems. The desired components can be composite such that they are created by integrating other components in the framework. In order to facilitate the composition of different applications, a CBAF-compliant framework provides a pool of software components encompassing a broad range of issues within the concerned domain. The frameworks based on the CBAF specifications are simple and flexible, imposing minimal restrictions and providing many of options for designing agents that will participate in a multiagent system.

3.1 Domain Characteristics

We use a component-based approach for our framework. Therefore, our framework implementations will be domain-oriented. At this point, we summarize some fundamental assumptions made about the type of agents supported by our framework, and in particular, the characteristics of domains for which we believe the methodology is appropriate:

- The application domain is assumed to be mature. We consider a domain as mature when there exist commonalities in its applications. We can then define these commonalities as domain-oriented components in our framework and reuse them in the different applications.

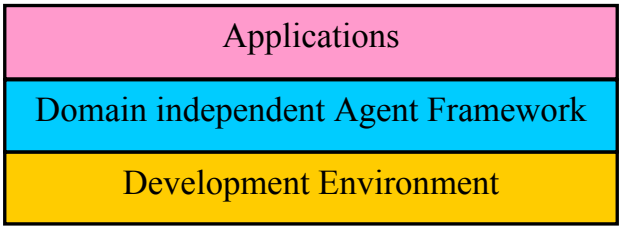
- The application domain is stable. Therefore, the domain is well defined in terms of agent roles, agent goals and communication mechanisms in its applications.

- Agents are heterogeneous, in that different agents may be implemented using different programming languages and agent architectures. The only constraint is that the development environment selected for agent implementation must support event-driven programming.

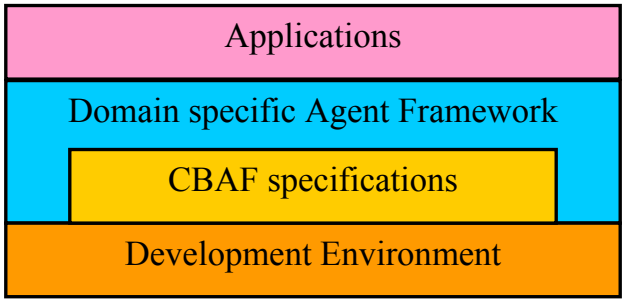
- The environment for the application domain is assumed to be dynamic, discrete, stochastic, episodic, and partially observable with real-time constraints.

3.2 CBAF Architecture

The CBAF architecture is a component-oriented approach for designing agent systems. CBAF defines the specifications for a designer to create an agent framework comprising a set of components that a user can utilize to develop sophisticated agent plans and associate the plans with individual agents. As mentioned before, this implementation will be domain-specific. Most of the current agent frameworks implement a specific agent system, and the users of these systems are expected to download the provided software and build their agents as extensions to the system. The CBAF approach differs from such frameworks in that it adds a layer of abstraction between the agent system implementation layer and the development environment. Although the agent framework implementation will be domain-specific, the learning curve for the users before they can start building application systems using the software becomes virtually non-existent.



Majority of Current Agent Frameworks



Component Based Agent Framework

Figure 3.1. Comparison of Agent Frameworks

Figure 3.1 compares CBAF with other current frameworks. CBAF does not provide any software libraries for agent applications; it just defines a set of rules for a *designer* to follow while developing agent frameworks, which a *user* can utilize for creating application systems. We show the usage, benefits, and applicability of the CBAF specifications by presenting an implementation of a soccer simulation system based on the CBAF architecture.

3.2.1 CBAF specifications:

The CBAF specifications defining the rules of a domain-dependent agent framework implementation that a designer and a user of the framework must adhere to are as follows:

1. An agent framework based on the CBAF architecture must be designed as a set of independent, self-contained, highly specialized components.

2. Commonalities of various applications in a domain must be discovered and developed as domain-specific components in the framework. Developers must also determine other generic and product-specific components for the domain. This approach is called *domain orientation* in CBSE [28] and it promotes software reuse and increased productivity during application development.

3. A component must expose its functionality through its interface. The interface must be complete and non-redundant, and it must hide the internals of the component. Since components are architecture-centric (depend on the component development context), multiple components with identical interfaces, but differing in implementations (one for each architecture to be supported in the framework) must be developed for the framework.

4. Components must be designed such that each performs a unique, singular function. Conversely, every function to be performed by an agent in a domain application must be defined as a component in the framework.

5. Framework designers must strive for the development of "context-free" components [28]. As far as possible, binding with particular contextual parameters such as data type, storage size, implementation algorithms, communication methods and operating environment must be postponed until component integration time when performance optimization is made.

6. The framework must facilitate the integration of its components in various ways to create different agent plans (also called as activities). It must also provide a way to compose these activities with the individual agents. Multiagent systems would be developed this way.

7. The agent framework must be designed for maximum flexibility in composing components. Towards that end, the framework must include the following types of components:

   - *Agent component(s):*

   The agent component represents an individual agent in the multiagent system. The component must be able to represent the internal states of the agent and the external states of the surrounding environment. The agent component must also receive sensor input from the environment and produce action output to the environment. The agent will be associated with one or more activity components representing the plans for the agent at design time. The scheduling mechanism for these activities and the internal control flow for the agent must be decided by the agent component. All variations of the above mentioned mechanisms must be represented as separate agent components in the framework.

   - *Activity component(s):*

   The activity component is an interface between an agent component and the user-defined activities or plans for the agent. The activity component must fully support the activity scheduling and agent communication mechanisms of the agent component. The activity component must also allow the creation of plans involving complex, decision-making routines. A plan must be rooted by an activity component and generated as a concatenation of rules based on the agent's internal state. The rules can have a positive or a negative classification. For each classification, a new set of

rules can be appended. This mechanism produces a data structure like a decision tree. Each branch of the tree structure must be terminated by an action to be performed by the agent when all the rules defined in the path are satisfied. The activity component must be able to solve the plan by propagating events through the plan. The agent framework can have one or more activity components satisfying all the above criteria.

- *Decision components:*

The decision components represent the rules used in plan generation for an agent. The rules can be defined as individual decision components or as a collection of the components chained together in some order. The framework must include all the decision components, so that the user is able to devise any rule that he wants to use in his plan. The commonalities in the various application systems developed in the concerned domain can serve as a guide to the required decision beans for the domain.

- *Behavior components:*

The behavior components represent the agent's actions. Each rule set in a plan has to be mapped to a behavior component. The framework must provide every possible atomic action for an agent in the system as a behavior bean.

Figure 3.2 describes a sample CBAF architecture for the RoboCup domain. The sample behavior and decision components shown in the figure are for the RoboCup domain.

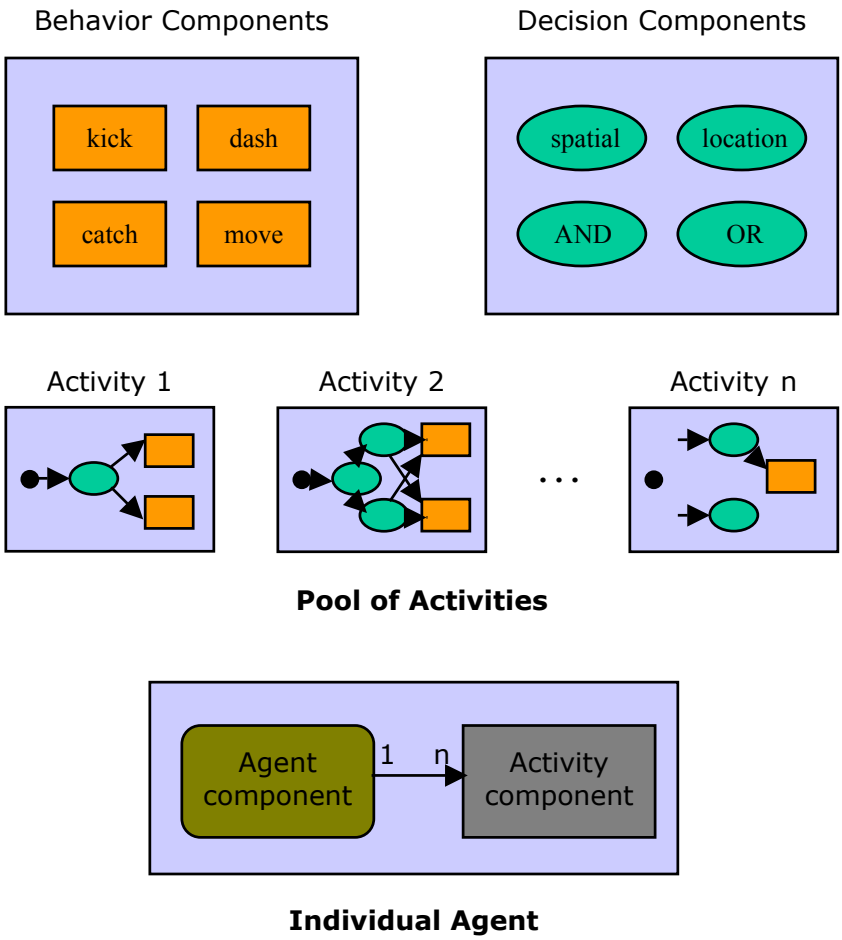


Figure 3.2. The CBAF architecture.

8. The agent framework must provide a look-up service for the agents. This feature might be built into the agent components in the framework. Both inter-agent and intra-agent communication must be facilitated through event triggering. The underlying architecture must also support communication with remote objects.

3.2 Challenges for CBAF design

We need to answer the following questions for a successful CBAF design:

- Which components should we develop in the agent framework for a given domain?

- The components have to be functionally independent, but they also rely on each other for accomplishing their tasks and hence should be interactive and cohesive. How do we achieve that?

- How do we achieve component integration to create new components?

- How do we deploy the whole system with multiple agents?

3.3 CBAF implementation with JavaBeans

The JavaBeans technology [11] by Sun Microsystems seems to be the most suitable development environment for creating agent systems using CBAF. Java supports the *delegation event model* for event handling, where an *event source* generates an event and sends it to a set of *event listeners*. The listeners must be registered with the source to receive notifications about specific event types. JavaBeans leverages the strengths of Java by providing a rich framework for manipulating events and the relationships between event sources and event listeners at design time. The Bean Development Kit (BDK) provided by Sun Microsystems for application development using beans presents a simple way of associating two beans with each other. Linking can be done by simply selecting the method that fires a particular event in the source bean and following it up with selecting the method that needs to be invoked on the listener bean each time that event is fired by the source. The BDK automatically creates the Adapter [7] class for

combining the source bean with the listener. Above all, the *introspection* and *reflection* mechanisms supported by Java and extended by JavaBeans open up a plethora of options for aiding complex decision-making by the agents. This paves the way for enabling the creation of a diverse set of applications using the system toolkit. Finally, with JavaBeans, the agent properties can be modified at run-time. This feature can be extremely useful while testing out different strategies for individual agents.

Among the currently available technologies, only JavaBeans provides all of the above-mentioned features. While other development environments like Visual Studio with Visual Basic or Visual C++ can also be used for developing components as DLLs (Dynamic Linked Libraries), they have limitations that add to the complexity of the task of agent system development. While these environments support event handling, combining the event source with the event listener is non-trivial. The Adapter class invoking the appropriate method in the listener component for each event generated by the source component has to be hand-coded. A similar case can also be presented for other distributed computing technologies like CORBA and COM. Since the above-mentioned technologies do not support introspection and reflection, the components will be highly specialized and inflexible. Hence, these technologies would need a much higher number of components to support complex decision-making than JavaBeans. Run-time modifications to agent properties would also be unavailable.

Chapter 4

SoccerBeans Framework

SoccerBeans is an agent framework for developing soccer teams in the RoboCup domain. SoccerBeans is aimed to serve as a prototype for agent framework development based on the proposed CBAF architecture.

4.1 Domain Analysis

In this section, we attempt to analyze the RoboCup domain by solving the challenges for CBAF design, as mentioned in Chapter 4. We have selected Sun's JavaBeans technology as the underlying architecture for our framework for the reasons mentioned in Chapter 4.

1. *Which components should we develop in the agent framework for a given domain?*

We need different Agent components for all the different architectures that we want to support in our framework. In SoccerBeans, we will support only one architecture, where the agents are built as JavaBeans (referred to from hereon as just *beans*). The activity

scheduling mechanism defined in [43] is used for these agents too. The PlayerFoundation bean is the Agent component in SoccerBeans.

Since we will support only one architecture, we will need only one Activity component. The Activity bean serves as the SoccerBeans' Activity component.

We will have to discover the commonalities and other agent-specific and application-specific decision criteria in the different agent plans developed for the various soccer teams, and use that information for determining the appropriate Decision beans for SoccerBeans. The most common decision criteria for soccer players are the issues related to its own and the ball's current position on the soccer field, and the current state of the game. We have defined 12 Decision beans for SoccerBeans. We describe them later in the chapter.

Some of the actions for a soccer player are dashing, kicking, turning, moving and setting the player's internal state. We have determined 10 Behavior beans for this domain, and these too are defined later.

2. *The components have to be functionally independent, but they also rely on each other for accomplishing their tasks and hence should be interactive and cohesive. How do we achieve that?*

Java supports the *delegation event model* for event handling, where an *event source* generates an event and sends it to a set of registered *event listeners*. The components that need to interact are the neighbors in an agent's design. For each connection, we have an event source and a registered event listener. We have defined two events to support the different messages that need to be communicated between the different sets of components. The PlayerFoundation bean sends the FunctionalityEvent message to its listener, the Activity bean, while the Activity bean and the Decision beans send out the ActivityEvent message to their listeners, other Decision beans and Behavior beans.

3. How do we achieve component integration to create new components?

The components designed in CBAF support reuse. For any player strategy that we want to design, we just have to select the appropriate components from SoccerBeans and link them properly. We demonstrate the design of a simple soccer player in Appendix B. The player is taught to dribble the ball to goal.

4. How do we deploy the whole system with multiple agents?

The UDP sockets handle the distributed programming issues for us in SoccerBeans. JavaBeans provides the BDK for deploying live components. Whether the individual agents in the multiagent system will function as a team or not will depend on how well

we have created them. We have developed a sample team using SoccerBeans. A short tutorial on executing the sample team is provided in Appendix C.

4.2 Components for the SoccerBeans Framework

The SoccerBeans framework consists of a pool of functional components developed as beans. The PlayerFoundation bean represents an individual player in the soccer system. The bean handles all the communication, knowledge representation, and activity scheduling aspects of the represented player. The framework provides decision beans that allow players to make decisions based on the current state of its world, and behavior beans that perform actions for the player. The Activity beans glue together the PlayerFoundation bean with the different player activities designed from the various decision and behavior beans provided by the framework. The following sections describe in detail the 24 beans that constitute the SoccerBeans framework.

4.2.1 PlayerFoundation bean

The PlayerFoundation bean encapsulates all the low-level details of the agent and allows the user to concentrate on the real issues of planning and coordination in a soccer team. The DatagramWrapper class handles the task of communicating with the soccer server. The player and server configuration information is static and is preserved in the ConfigurationData class of the bean. The player receives information about its surrounding environment from the soccer server at every simulation step. The

information concerns with the relative positions of the static (for e.g., flags on the field) and dynamic (for e.g., the ball and other players) objects on the field and it is preserved in the player's WorldModel. The bean has a reference to the RobocupUtilities class, which is a library of utility functions defining the various decisions and actions for a player. In conjunction with the Activity bean, the PlayerFoundation bean provides for the player an activity scheduling mechanism that supports both the BDI and the Subsumption architectures. Figure 4.1 describes the algorithm.

```
input = the new input
activities = set of all player activities
matches = new Vector()
for all i in activities do
  if i.canHandle(input) then
        matches.addElement(i)
  end if
end for
uninhibited = new Vector()
for all i in matches do
  inhibited = false
  for all j ≠ i in matches do
        if j.inhibits(i) then
          inhibited = true
        end if
  end for
  if not inhibited then
        uninhibited.addElement(i)
  end if
end for
chosen = random element from uninhibited
chosen.handle()
```

Figure 4.1. The PlayerFoundation bean's activity scheduling algorithm

The PlayerFoundationBeanInfo class exposes many properties of the PlayerFoundation bean. These properties differentiate the players from each other and

can be manipulated by the user at design time and, if required, also at run time. The name and team properties define the player's name and team. A value for playerNumber is assigned to the player by the server when the initial connection is established. The player's starting position before kickoff can be set by the initialLocation property. Rectangular zone area can be specified for the player through the zone property. This allows the player to make special case decisions when a particular dynamic object is in its zone area. Age defines the number of cycles for which the ghost of a dynamic object is preserved in the world model. This feature allows the player to remember a dynamic object that has recently moved out of its view range, but high age values can result in noise in the world model. The display property pops up a new window that graphically displays the player's world model. This can be very useful while debugging. If the debugFileName is not empty, then the PlayerFoundation bean spits out the debug information into that file. The file defining the player and server configuration information must be entered in the configFileName field. Hostname and portNumber for the soccer server must be defined in the corresponding fields for the bean. The version information is used in establishing the initial connection with the server. This value must be set to 5.00 for the SoccerBeans system. If the player is a goalie, then the goalie property must be set to true. The player must be set online only after all the other properties are defined and all the activities are added to it. This initiates the socket communication between the server and the player.

As defined by PlayerFoundationBeanInfo, the PlayerFoundation bean acts as an event source for the ActivityEvent events. The bean provides methods for adding and

removing the listeners. The ActivityEvent events are fired to pass references of the source bean and the current input to the listeners. The Activity beans are the recipients of these event notifications in the SoccerBeans system. While firing these events, the PlayerFoundation bean can invoke either of the addActivity, canHandle or handle methods on the listener bean. The addActivity method is invoked on an activity at the instant when the activity is registered with the player at design time. The references of both the activity and the player are exchanged at that instant. The canHandle method determines if the concerned activity can handle the current input to the player. As part of the PlayerFoundation bean's activity scheduling mechanism, the canHandle method is invoked for all registered activities at every instant when a new input is received by the player from the server. The handle method then executes the concerned activity. Exactly one matching, uninhibited activity is selected for execution at each simulation step.

The PlayerFoundation bean is not a recipient of any event notification and hence does not expose any methods for invocation.

4.2.2 Activity bean

The design of every new plan or activity for an agent begins with the Activity bean. The activities are generated as sets of rules with each branch of the resulting decision tree terminated by an action to be performed by the agent. Thus, if a particular set of rules defined in a path are valid for a player and its surrounding environment then the player performs the action defined at the end of that path. The rules are designed by combining

the various decision beans, while the actions performed are defined by the behavior beans. The root of each such path is an Activity bean.

The ActivityBeanInfo class exposes two properties for the Activity bean, the name of the activity, and the inhibits property. The list of other activities that are subsumed by this activity must be declared in the inhibits property. The activities must be referred to by their name, and delimited by commas.

The Activity bean acts as glue for associating a player to one of its activities. The bean is an ActionEvent event listener and implements the addActivity, canHandle and handle methods declared in the ActivityListener interface. These methods are exposed for invocation by the PlayerFoundation beans through the ActivityBeanInfo class. Each activity comprises two decision trees. The tree associated with the canHandleListener in the Activity bean determines if the current environment settings are favorable for performing the activity. The other tree is associated with the handleListener in the Activity bean and defines how the activity must be executed. The first tree is solved by the canHandle method, while the second tree is executed by the handle method.

The decision trees in the Activity bean comprise some decision and behavior beans linked with each other in some sequence. For both methods, canHandle and handle, the Activity bean provides add and remove methods for decision and behavior beans. The ActivityBeanInfo class exposes these methods. The structure of both the methods

for solving the trees is identical. A FunctionalityEvent event defining the current state of the world is fired from the method and the notification is sent to the decision or behavior bean adjacent to the Activity bean in the chain. Both DecisionListener and BehaviorListener interfaces listen to FunctionalityEvent events and are implemented by the decision beans and the behavior beans respectively. If the next bean in the chain is a decision bean, the Activity bean invokes the decide method on the decision bean. If it is a behavior bean, then the behave method is invoked. The notification is propagated further in the appropriate path by the decision beans until it hits the behavior bean where it is terminated. For canHandle, the behavior bean sets the canHandle flag in the Activity bean either directly or via an intermediate decision bean. This determines whether the activity can be performed in the current cycle with the given environment. If the activity is applicable then, as part of the SoccerBeans system's scheduling mechanism, all other activities applicable for the current cycle that are inhibited by this activity are eliminated. The activity that is handled for the current cycle is selected from this new list of applicable plans. For handle, the behavior bean typically sends a message to the soccer server describing the player's action for the current cycle.

4.2.3 Decision beans

The decision beans enable a user to define rules for generating plans. Every possible condition that the user might need to check for making his/her decision must be encompassed by the published set of decision beans in the framework. The user must be able to define any rule by using either a single decision bean or combining a collection of

them in some form. Plans are generated by chaining such rules with an Activity bean at the head and a behavior bean at the tail.

For every path in a plan, a decision bean is preceded by either an Activity bean or another decision bean, and is followed by a behavior bean or another decision bean. Whenever a decision has to be made, a message has to be propagated through a chain from the Activity bean towards the behavior bean via the intermediate decision beans. This is achieved in SoccerBeans by sending FunctionalityEvent event notifications through the chain. The event object provides the listener with a picture of the agent's current internal state. Every decision bean implements the DecisionListener interface for listening to the FunctionalityEvent event notifications. The decide method for each decision bean performs the necessary computation and sets the decision flag appropriately. The path to be pursued is selected based on the value of the flag, and the event notification is propagated further in that path.

As discussed above, a decision bean can be linked to either another decision bean or a behavior bean. Also, two different chains of beans will be connected to the decision bean, one for each possible value of the decision flag. For both chains, a decision bean provides add and remove listener methods for connecting to both types of beans.

The SoccerBeans framework provides the following decision beans:

4.2.3.1 DInputType bean

The input received from the soccer server is represented as SensorInput, while the action event generated by the PlayerFoundation bean to elicit an action for the current cycle is represented as Event input in the SoccerBeans system. The DInputType decision bean can be used to classify player behaviors based on the type of input propagated by the FunctionalityEvent. The DInputTypeBeanInfo class exposes the inputType property of the DInputType bean. The valid entries for the property are SensorInput and Event.

4.2.3.2 DClosePlayers bean

The DClosePlayers bean can be used to create rules based on the number of players within some distance to the player represented by the PlayerFoundation bean. The players considered can be from either team or irrespective of the team. A user can also set the considered distance and the number of players considered at design time. The DClosePlayersBeanInfo class presents the users with the players, distance and number properties for the above functions. The players property can be set to Teammates, Opponents, or Both. The number property must be set to some positive integer, while distance must be a positive double value. The decision flag is set to true if the number of players counted is greater than the value of the number property, else it is set to false.

4.2.3.3 DClosestToBall bean

The DClosestToBall bean facilitates decision-making based on whether the player is closest to the ball amongst all the other players. The other players can either be just

teammates or both teammates and opponents. The onlyTeammates flag determines the selection, and it is the only user modifiable property of this bean published by the DClosestToBallBeanInfo class.

4.2.3.4 DBallDistance bean

The DBallDistance bean can be used for classifications based on the distance of the ball from the player. The ball distance can be compared either to a specific number, or to a particular field in the server's configuration file. A user can also set the comparison operator. The DBallDistanceBeanInfo class exposes the properties relation, field, configField, and distance for the DBallDistance bean. The bean supports the following operations for the relation property: <, >, =, <=, >=, !=. The field flag determines whether to consider the specified server configuration parameter or the distance property for comparison. All the parameters stored in the ConfigurationData class in the framework are valid entries for the configField property. The distance property allows the user to specify the threshold distance. The actual distance is determined from the player's world model.

4.2.3.5 DSelfDistance bean

The DSelfDistance bean allows classifications based on the player's distance from some static location on the field. The location can be specified either as a point in Cartesian coordinates, or as a static point (for e.g., flags) represented in the world model. The comparison operator can also be set as in the DBallDistance bean. The DSelfDistanceBeanInfo class presents the location, relation, and distance properties of this bean for user modification.

4.2.3.6 DBallPosition bean

One of the most frequently used criterion for deciding on the next action for a player is the ball's current location on the soccer field. The DBallPosition bean allows the player to make such decisions. The bean has four properties, coordinateX, coordinateY, relationX, and relationY as defined in the DBallPositionBeanInfo class. The properties correspond to the X and Y coordinate values for the ball, and the comparison operation to be considered by the bean for the X and Y coordinates. The coordinateX and coordinateY properties can hold a double value or *, while the relationX and relationY properties can be set to any of <, >, =, <=, >=, or !=. If the coordinate value is *, then the corresponding coordinate is not considered for comparison. So, consider the case where the bean's property values are set as follows: coordinateX = 25, coordinateY = *, relationX = <, and relationY = >. Here, the decision flag will be set to true if the X coordinate for the ball's current location is less than 25, and false otherwise. The ball's position along the Y axis is ignored.

4.2.3.7 DSelfPosition bean

The DSelfPosition bean is identical to the DBallPosition bean, except that this bean considers the player's current position on the field in lieu of the ball's position. The DSelfPositionBeanInfo class also defines the coordinateX, coordinateY, relationX, and relationY properties for this bean. So, if the bean's properties are set as coordinateX = -45, coordinateY = 5, relationX = >=, and relationY = <, then the bean's decision flag will be set to true only if the player's current position along the X axis is greater than or equal to -45 and its position along the Y axis is less than 5.

4.2.3.8 DZonalPosition bean

As mentioned in the PlayerFoundation bean, each player has its own zone area on the soccer field. Additional zones are defined in the WorldModel class and are common for all the players. The DZonalPosition bean can be used to enable decision-making based on the presence (or absence) of either the ball or the player itself in a particular zone on the field. This bean has two user modifiable properties, zone and object, that hold the zonal and object information respectively for consideration in the current rule. The valid entries for the zone property are all the zones defined in the framework. Some of the zones commonly used in the sample team are – PlayerZone (the player's zone area), DefensiveArea (the player's half), OffensiveArea (the opponent's half), TopArea (the top half), BottomHalf (the bottom half), DefensiveAlertArea (the alert area in the player's half), DefensiveHighAlertArea (the high alert area in the player's half), OffensiveAlertArea (the alert area in the opponent's half), and OffensiveHighAlertArea (the high alert area in the opponent's half). Refer to the source code for the WorldModel class for further information about zones. The valid entries for the object property are ball, self and playerX, where X is the uniform number of the teammate. The bean's decision flag is set to true only if the specified object is currently present in the specified zone. The bean uses Reflection APIs for invoking the appropriate method in the appropriate class for the specified zone.

4.2.3.9 DBallUnknown bean

The DBallUnknown bean allows the player to make a decision based on whether it has seen the ball recently or not. This bean has no properties for the user as it simply checks

whether the player knows the current location of the ball in the world model. The decision flag is set to true if the player has *not* seen the ball recently, and false otherwise.

4.2.3.10 DPlayMode bean

The current play mode, as set by the soccer server, can be used as the criterion for classification by using the DPlayMode bean. The DPlayModeBeanInfo class exposes the playMode property of the bean, which can be set to all the valid play modes for the soccer system.

4.2.3.11 DExecuteMethod bean

The SoccerBeans framework provides many decision beans, covering a wide range of potential decision criteria for players. However, there are bound to be cases where a particular capability required by a user is not accounted for in any of the decision beans in the framework. For such cases, users have to extend the SoccerBeans framework. The framework can be extended either by designing new beans that provide the missing capabilities, or by defining new utility functions that handle the required capabilities and invoking the routines with the DExecuteMethod bean. The DExecuteMethodBeanInfo class provides the user with the opportunity to define such a method along with the class defining the method and the input parameters for the method by exposing the className, methodName, and methodParams properties of this bean. The method parameters must be delimited by commas, and each parameter must be defined as the input data type followed by the input value, separated by a colon. (This syntax for method parameters is common for all beans in the framework, and will

not be mentioned from now onwards in the documentation). The post-condition for such a user-defined utility function must be that it returns a Boolean data type. The decision flag for this bean is set to the return value of the routine. A sample user-defined utility function is the testDExecuteMethod() method defined in the RobocupUtilities class. The bean uses Reflection APIs for invoking the specified method with the provided parameters in the specified class.

4.2.3.12 DIfThenElse bean

The DIfThenElse bean is defined for classification based on a rule for which there is no explicitly defined decision bean. The DIfThenElse bean can be linked to any behavior bean or a chain of decision beans terminated by a behavior bean for defining the rule. The DIfThenElse bean passes the FunctionalityEvent event notification to the linked bean for determining the decision. The terminating behavior bean must set the decision flag of the DIfThenElse bean to true or false. An additional set of add and remove listener methods for the new chain.

P.S. The DExecuteMethod bean provides a more pragmatic solution to the above problem and it should be preferred in lieu of the DIfThenElse bean.

4.2.4 Behavior beans

As described in the previous sections, plans are generated as sets of rules, with each branch of the resulting tree structure terminated by an action to be performed by the agent. These actions are defined as behavior beans. The actions typically include setting a particular property of some object or executing a particular method for some object. The

agent framework must comprise all the behavior beans such that a user is provided with the opportunity to perform any action deemed suitable for his/her plan.

The behavior beans are preceded by either an Activity bean or a decision bean. Whenever a decision is to be made in SoccerBeans, a behavior bean receives a FunctionalityEvent event notification from the preceding bean. Every behavior bean is defined as a listener of these events by implementing the BehaviorListener interface. The behave method for each behavior bean performs the specified action for the bean.

The SoccerBeans framework provides the following behavior beans:

4.2.4.1 BBoolean bean

The BBoolean bean can be used for setting a particular boolean property of an object to the specified value. The BBooleanBeanInfo class exposes the className, propertyName and value fields of the BBoolean bean. These fields can be used to specify the property that needs to be modified with its new value and the class defining the property. The bean uses Java's Reflection API to locate the set method for the property in the specified object and invokes the method, passing the specified value as the parameter. Thus, any boolean variable in the system can be modified at run time by using this bean.

4.2.4.2 BIncorporateObservation bean

The BIncorporateObservation bean is meant for receiving the sensor inputs from the soccer server and incorporating them into the player's world model. There is no property in the bean for a user to modify and the BIncorporateObservationBeanInfo class is defined accordingly.

4.2.4.3 BDash bean

A player can dash to a particular point on the soccer field by using the BDash bean. The position to be dashed to can be specified either as a point in Cartesian coordinates or as a location derived by executing some method in the framework. The bean currently supports method invocations for determining the current locations of the ball, and of any static flag on the field represented in the world model. Users might want to have their own routines for computing the position that a player should dash to given the current state of its world. These routines are also accommodated in the BDash bean. This bean has seven user modifiable properties – location, method, className, methodName, methodParams, maxPower, and tolerance, as published by the BDashBeanInfo class. The location property holds the destination point in Cartesian coordinates. The X and Y coordinates must be delimited by a comma, and can be specified either as double values or as simple expressions. The supported expressions are simple functions with respect to the ball's or the player's current position, such as ballX + 13.5, selfX – 4.3, ballY * 1.5, and selfY / 3. The method flag determines whether the player must derive the destination point from the location field or by executing the specified method. The className, methodName, and

methodParams are used for specifying the method to be invoked for the bean. The maximum power with which the player will dash to the specified location will depend on the maxPower field. Lower maxPower values might be considered for relatively insignificant dashes to preserve the player's stamina for useful work. The BDash bean's tolerance property smoothens the player's movements upon reaching close enough to the destination point by eliminating unnecessary dashes and turns. The player assumes to have reached its destination when it is within the distance specified by the tolerance property from the destination. The tolerance value must be greater than 2.

4.2.4.4 BDribble bean

The BDribble bean allows the player to dribble the ball to a specified point on the field. Just like for the BDash bean, the point for this bean can also be defined either in Cartesian coordinates or by executing a method. The BDribbleBeanInfo class publishes the following properties of this bean – finalPosition, method, className, methodName, methodParams. All the properties in BDribble are identical to the corresponding properties in BDash. (The finalPosition property is similar to BDash's location property).

4.2.4.5 BShoot bean

The BShoot bean is similar to the BDribble bean except that while the latter dribbles the ball, this bean kicks it with maximum force. The BShootBeanInfo class provides similar options to the user as the BDribbleBeanInfo class.

4.2.4.6 BKick bean

The BKick bean allows the player to kick the ball to a specified position. This bean is deprecated and its functionality is better served by the BDribble and BShoot beans.

4.2.4.7 BMove bean

A player can move to a particular position on the field with the help of this bean. The BMoveBeanInfo class provides the coordinateX and coordinateY properties for this bean.

4.2.4.8 BTurn bean

The BTurn bean enables the player to turn by a specified angle. A user can define the angle either explicitly as a double value, or implicitly by specifying the object that the player must face in the next simulation step. The user is presented with three modifiable properties for BTurn – angle, considerObject, and object. The angle property holds the value that the player must turn by, the object property holds the name of the object that the player must face, and the considerObject flag determines whether to consider the object or the angle. The valid entries for the object property are ball, any static flag, and playerX, where X is the uniform number of the teammate.

4.2.4.9 BCatchBall bean

This bean is for the goalie to catch the ball inside the penalty area. The BCatchBall bean does not have any properties for the user.

4.2.4.10 BExecuteMethod bean

Just like the DExecuteMethod bean, this bean is to accommodate the player actions that are not handled by any behavior bean in the SoccerBeans framework. The BExecuteBeanInfo class exposes the className, methodName, and methodParams properties for the user-defined routines. The testBExecuteMethod() method in the RobocupUtilities class is a sample routine that can be used with this bean. The BExecuteMethod bean can be especially useful for implementing set plays for players.

Chapter 5

Testing and Results

Figure 5.1. The sample team

In order to test the software, we created a soccer team using the framework. In this chapter, we describe the 11 agents that we designed for our soccer team. We also provide some test results. The documentation on running the sample team is provided in Appendix C of this report. We also provide a tutorial in Appendix B that explains how to

use the SoccerBeans framework. We design a simple player that can dribble the ball to goal in the tutorial.

5.1 The sample team

Each player in the sample team has several activities associated with itself. All the players share two of these activities – *Observe* and *StartPlay*.

- *Observe activity:*

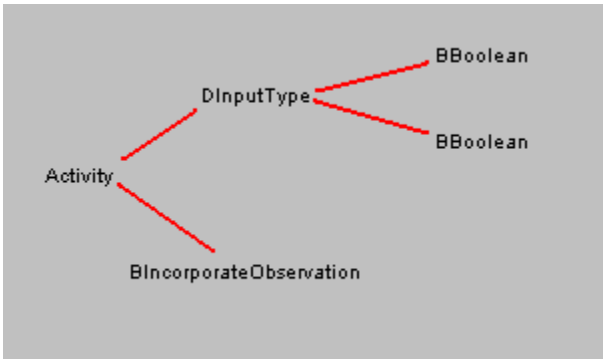


Figure 5.2. Observe activity

This activity enables the associated player to receive fresh inputs from the soccer server about its surrounding environment. These sensory inputs are incorporated into the player's world model. *Observe* is set to inhibit every other activity for all the players.

- *StartPlay activity:*

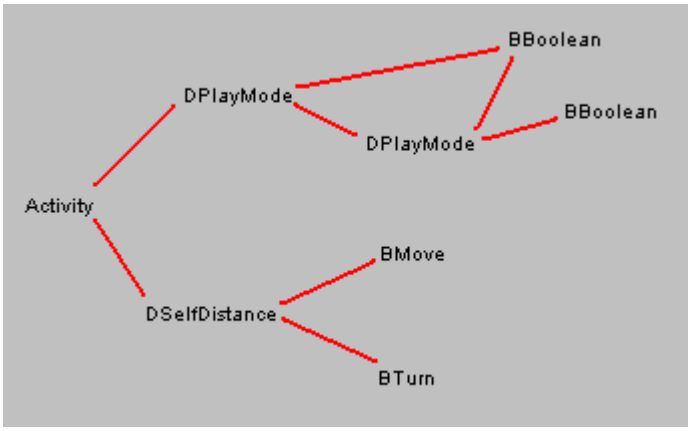


Figure 5.3. StartPlay activity

This activity resets the player's position to its pre-defined initial position after a goal or before kick-off. The initial positions for the different players are shown in their respective sub-sections. The player is set to face towards the ball after repositioning.

We explain the design strategies for each player in the rest of this sub-section.

5.1.1 Goalie

Besides the above-mentioned plans, our goalie can perform the following activities:

- *GoalKick activity:*

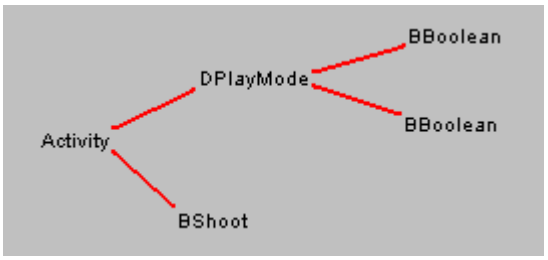


Figure 5.4. GoalKick activity for goalie

Only the goalie can kick the ball when the play mode is goal_kick. This activity handles this particular scenario for the goalie. The goalie dashes to the goal and kicks it towards the center of the field.

- *Catch activity:*

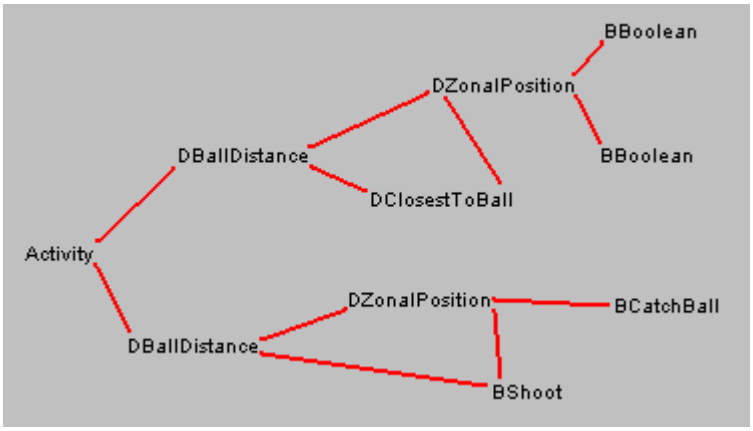


Figure 5.5. Catch activity for goalie

The *Catch* activity enables the goalie to catch the ball when it is in the penalty area. This activity is handled if the ball is in the defensive high alert zone as defined in the world model, and the goalie is either the closest team member to the ball or within 10 meters from the ball. The goalie catches the ball if it is at a catchable distance; otherwise, it kicks the ball towards the center of the field.

- *Pass activity:*

Once the goalie catches the ball, it moves to the corner of the penalty box that is on the other half of its current position through this activity. The condition that the goalie has caught the ball is tested by the ball distance (whether it is less than the catchable distance) and the current play mode (the play mode changes to free_kick when the goalie catches

the ball). When activated, the goalie moves to the appropriate position, and kicks the ball towards the center of the field after that.

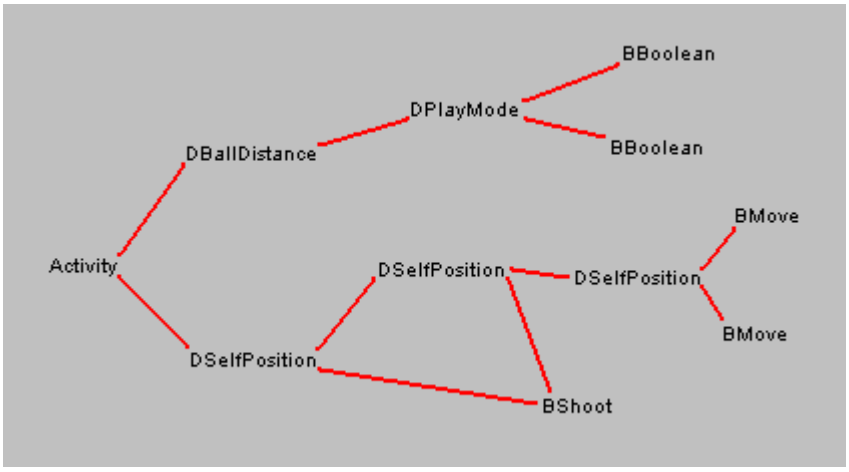


Figure 5.6. Pass activity for goalie

- *GotoPosition activity:*

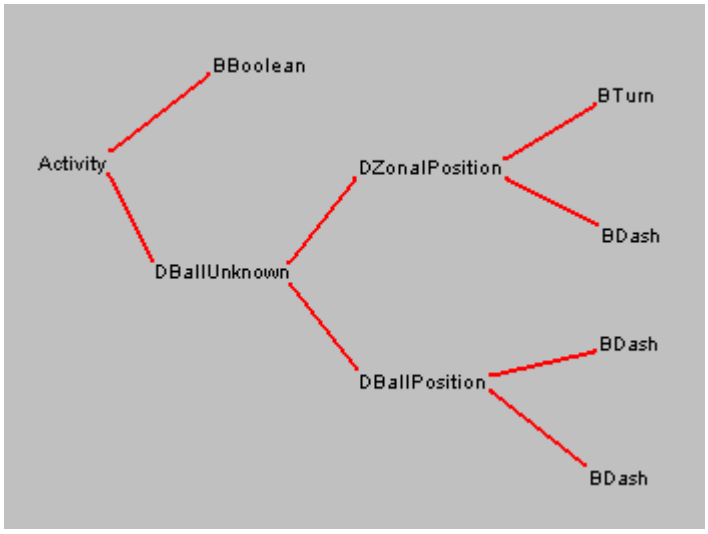


Figure 5.7. GotoPosition activity for goalie

This is the goalie's default activity, executed when none of the other activities can be handled. This activity enables the goalie to position itself at an appropriate location with respect to the ball.

5.1.2 Defenders

The two defenders have similar activities and are, therefore, presented together. These are other activities associated with the defenders:

- *Pass activity:*

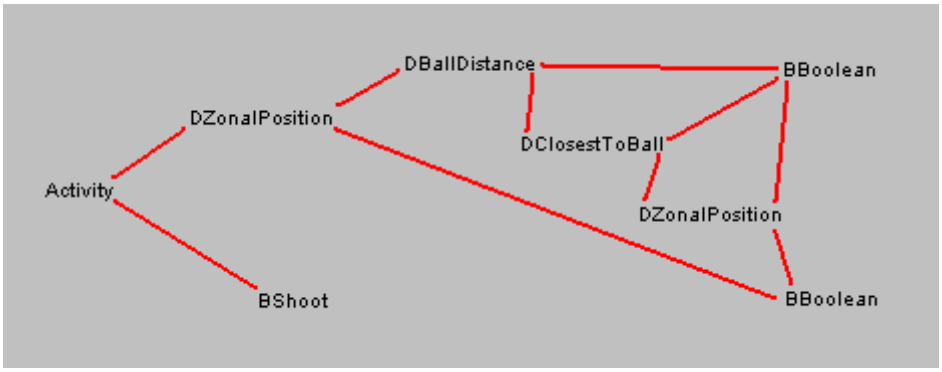


Figure 5.8. Pass activity for defenders

This activity is executed when either of the following conditions holds: the ball is less than 10 meters away, the defender is the closest team member to the ball, or the ball is in the defensive high alert area. The player must also be in its zone area, else it will chase the ball well beyond its home base. If *Pass* is executed, then the defender will dash to the ball and kick it with maximum force towards the opposite side.

- *GotoPosition activity:*

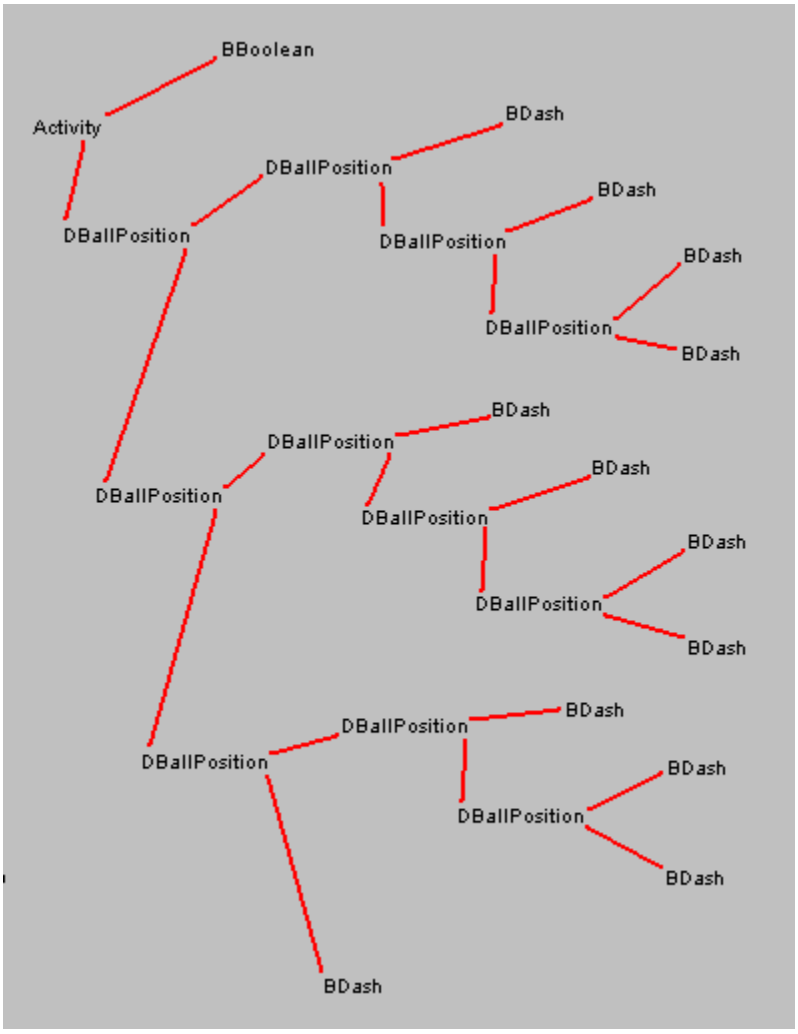


Figure 5.9. GotoPosition activity for defenders

This is the defender's default activity that positions itself with respect to the ball.

5.1.3 Defensive Midfielders

The three defensive midfielders in our sample team can perform the following activities in addition to *Observe* and *StartPlay*:

- *Dribble activity:*

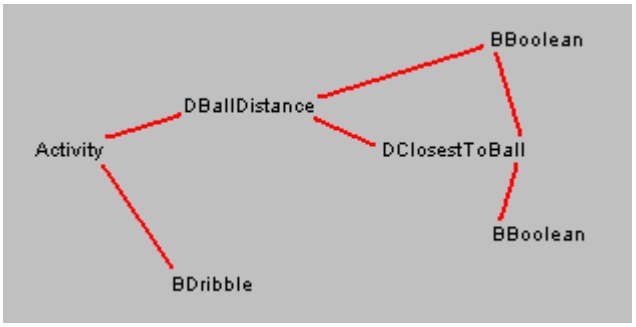


Figure 5.10. Dribble activity for defensive midfielders

The player dribbles the ball parallel to the X axis from its current position with this activity. It is activated if the ball is inside 10 meters from the player's current location, or if the player is the closest team member to the ball.

- *Pass activity:*

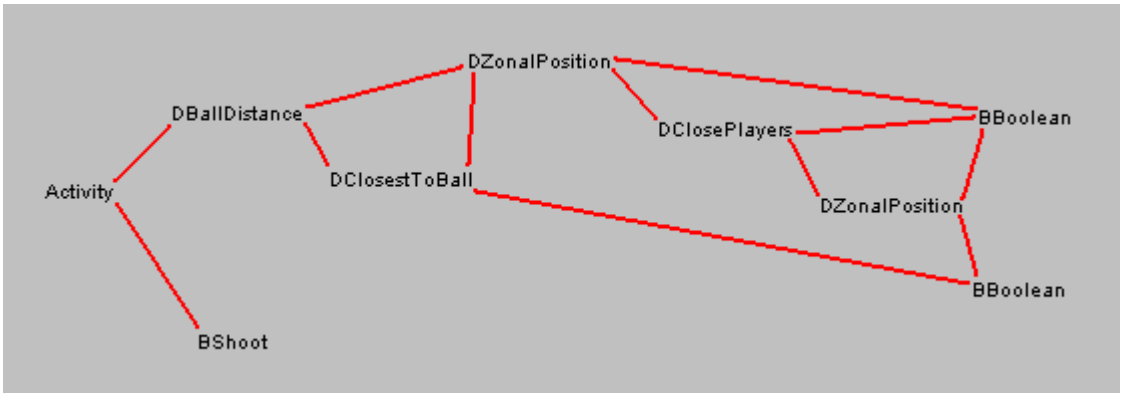


Figure 5.11. Pass activity for defensive midfielders

The *Pass* activity is executed for the player when all conditions for *Dribble* hold, and at least one of the following holds: the ball is either in the defensive high alert area or in the player's zone area, or at least one opponent is too close to the ball for comfort. This activity subsumes the *Dribble* activity.

- *GotoPosition activity:*

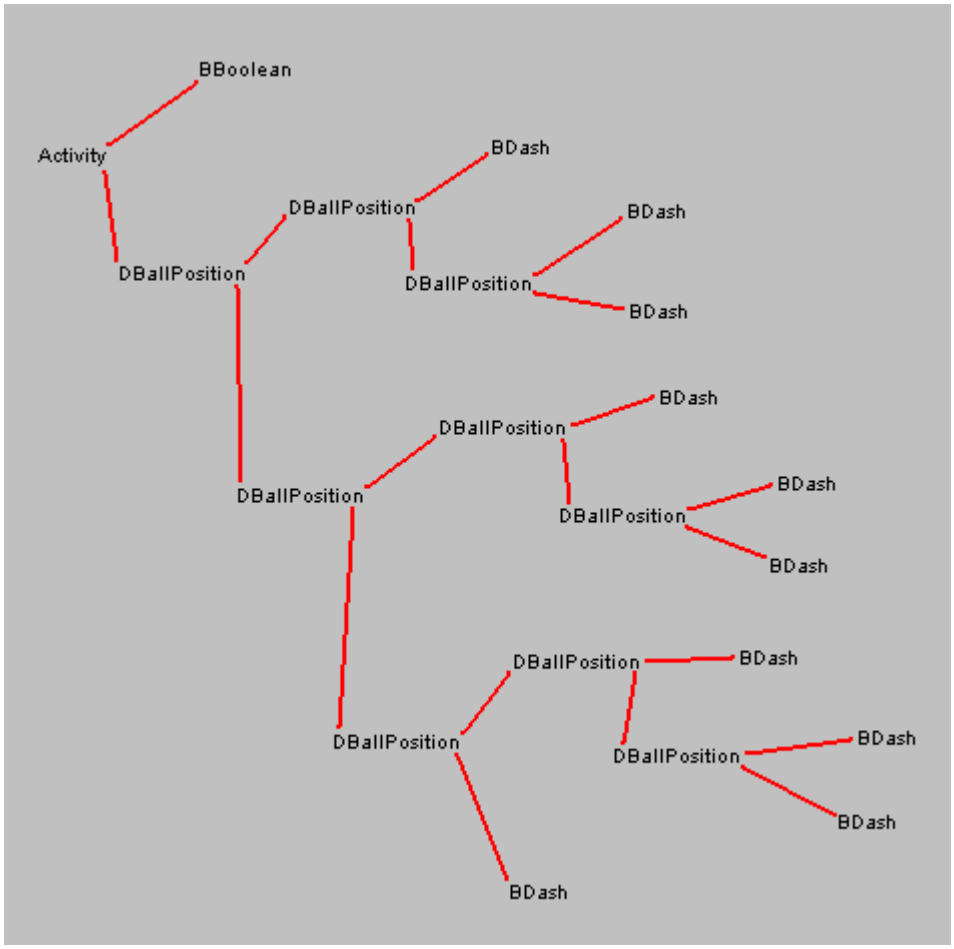


Figure 5.12. GotoPosition activity for defensive midfielders

This is the default behavior for a defensive midfielder and it positions the player to the most suitable location for tackling the ball. The defenders and the defensive midfielders show a tendency to converge on the ball as the ball gets deeper into the defensive alert zone.

5.1.4 Offensive players

In addition to *Observe* and *StartPlay*, the offensive players also share the following *Dribble* and *Pass* activities:

- *Dribble activity:*

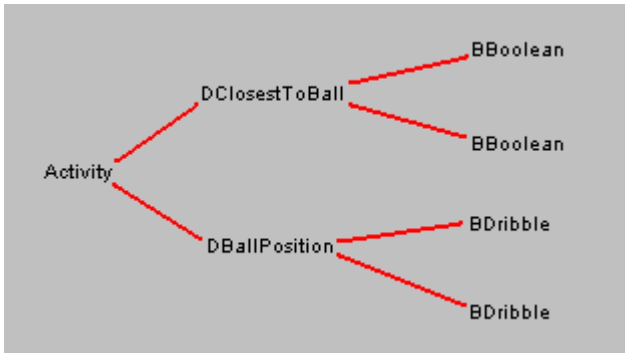


Figure 5.13. Dribble activity for offensive players

An offensive player dashes towards the ball only if it is the closest team member to the ball. Through this activity, the player dribbles the ball to the opposition goal post if the ball is in the offensive alert area, otherwise the ball is dribbled parallel to the X-axis from its current location.

- *Pass activity:*

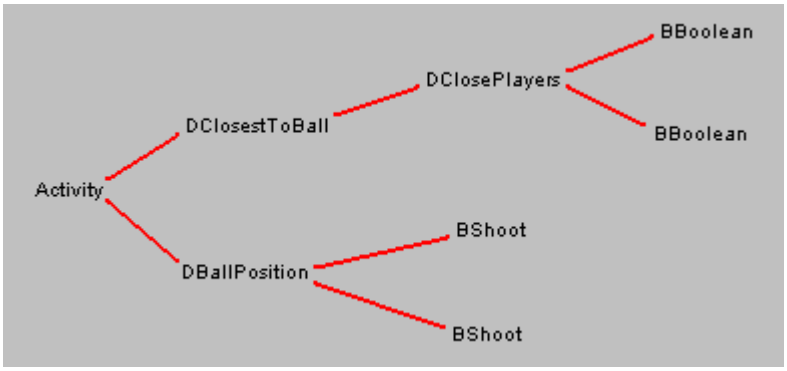


Figure 5.14. Pass activity for offensive players

This activity is executed if the offensive player is not only closest team member to the ball, but also has opposition players closing in. While executing this activity, the player kicks the ball towards the opposition goal post if the ball is in the offensive alert area, otherwise it is kicked straight and deep into the opposition half. *Pass* subsumes *Dribble*.

- *GotoPosition activity:*

The offensive players differ from each other in their *GotoPosition* activities. The offensive players show a tendency to repel each other, thereby providing more options for passing.

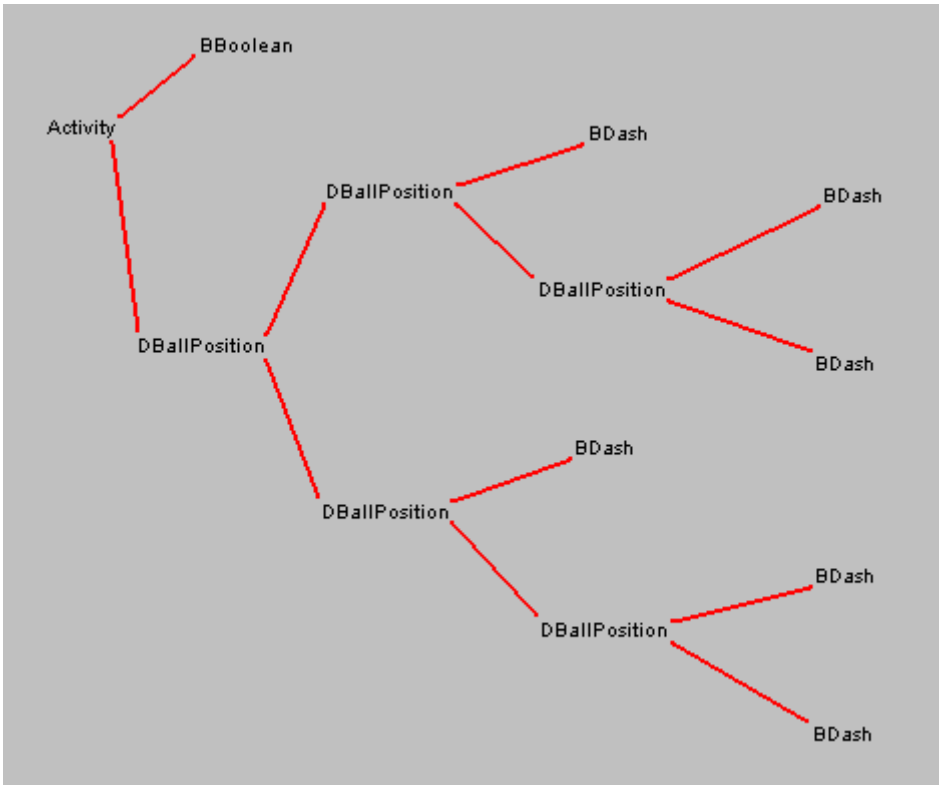


Figure 5.15. GotoPosition activity for offensive center midfielder

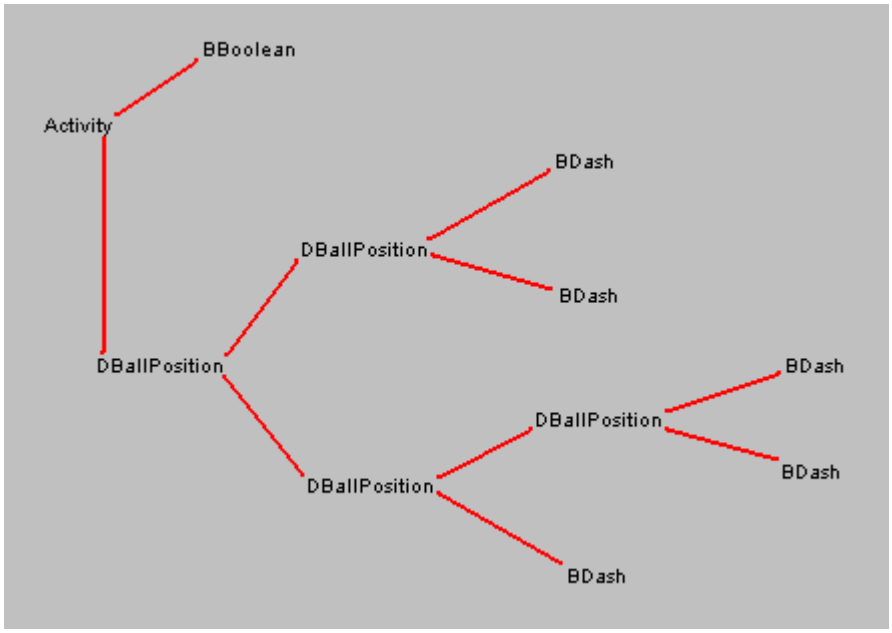


Figure 5.16. GotoPosition activity for other offensive midfielders

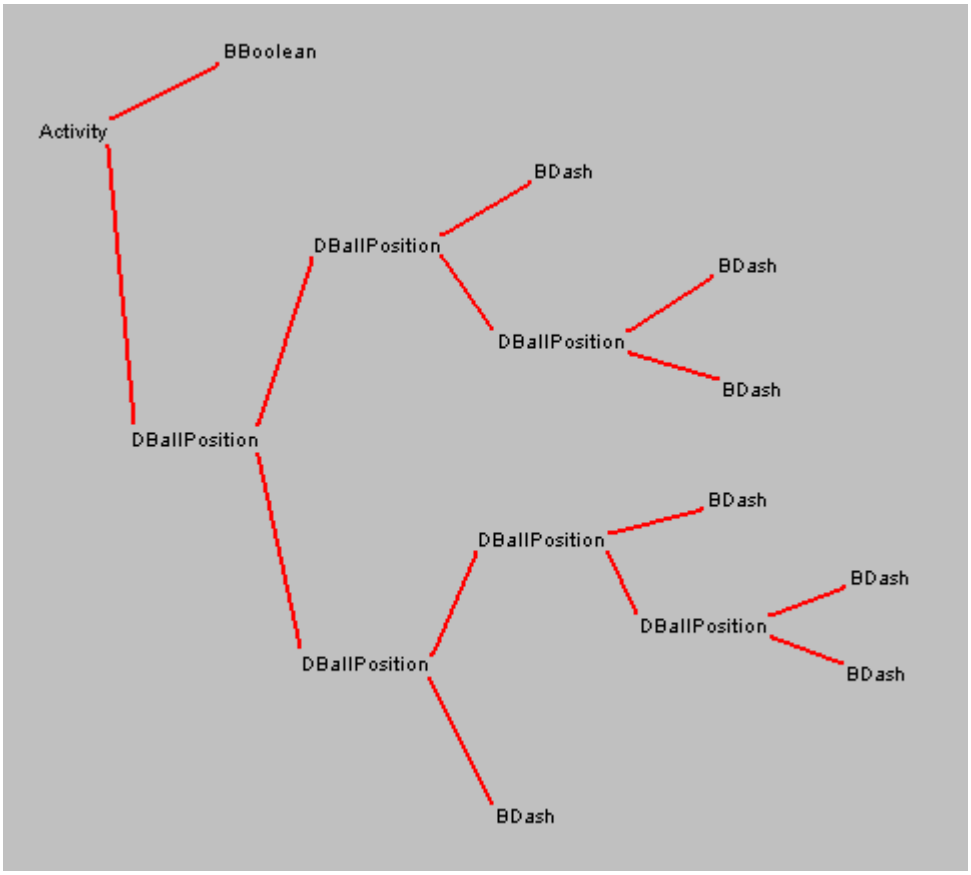


Figure 5.17. GotoPosition activity for offensive players

5.2 Comparisons and Results

Figure 5.18. Gamecocks vs. Chikomalos

The sample team, named Gamecocks, was tested against Chikomalos, the best soccer team we have using Biter. The final score after a playtime of 10 minutes was Gamecocks 18, Chikomalos 0.

Chapter 6

Conclusions and Future Directions

6.1 Conclusions

The objective of our research work was to discover a new approach for designing multiagent systems that would facilitate rapid application development without sacrificing for performance or functionality. Our work has been a success towards that end, albeit it is suitable for only a subset of real-world applications.

We have introduced our Component-Based Agent Framework (CBAF) architecture for developing agent frameworks, along with SoccerBeans – a CBAF implementation for the RoboCup domain. The CBAF architecture attempts to leverage the multiagent system development methods by taking advantage of the research done in component-based software engineering. We define our CBAF specifications for developing agent frameworks and using these frameworks for developing application systems based on the CBAF architecture. A CBAF-compliant framework is unique in that there is virtually no learning curve for its users before they can start developing multiagent systems using the framework. In addition, the integration of the agent architecture with a component-based architecture like JavaBeans reduces the task of creating agents to that of visually linking

the agents to its activities. These activities are also created by visually linking the necessary building blocks from a pool of available components. Our system allows its users to quickly design and deploy their agents without worrying about the underlying architecture.

In contrast to the popular agent frameworks like Zeus and JADE, a CBAF-compliant framework is domain-specific. It is specialized for a particular application domain, and this allows it to provide users with features like multiagent system creation through just drag and drop of components. Similar to Zeus, CBAF is also aimed at enabling users with only basic competence in agent technology to create functional multiagent systems. Zeus provides a user with tools to define the agents and the tasks (or goals) that they need to accomplish along with the pre-conditions and the post-conditions for each task. Given a goal, the agents in Zeus generate plans to achieve the goal. However, the user is expected to supply the code that implements the tasks.

We have demonstrated the ease and the usefulness of CBAF specifications with our SoccerBeans implementation for the simulated soccer application. We analyzed the RoboCup domain and described the components developed for the framework. We also demonstrated the usability of SoccerBeans by constructing a sample team of soccer playing agents. The sample team was tested against the best available soccer team using Biter, and the sample team won 18-0. While the final score in a game can only be viewed with a grain of salt while concluding about efficiency of the software, it certainly shows that performance was not compromised in SoccerBeans in order to increase its flexibility.

When we consider the other advantages that SoccerBeans offers us, we can certainly conclude that it is a big step forward from its predecessor. SoccerBeans is being used for the first time this semester in the Multiagent Systems class at our University, and the student reviews will provide further information for analyzing the framework.

As mentioned earlier, the CBAF approach for developing agent frameworks is not suitable for every real-world multiagent application domain. Since component-orientation plays a major role in this method, only the application domains that support componentization should be considered. As we describe in Chapter 3, these application domains are mature and stable, and only such domains will have some common, well-defined characteristics reused amongst multiple agents and application systems for the domain that our architecture can exploit.

6.2 Future Directions

To date, we have tested the CBAF architecture only for the RoboCup domain. The architecture can be tested with other domains like the trading agent problem, resource management, and supply chain management in the future. How can we develop a CBAF-compliant framework that is also FIPA-compliant? It would be worthy to find an answer.

Regarding SoccerBeans, the software can be extended in a few ways. The current decision beans support agent decisions based on the ball's current position, or the agent's own position. No beans enable an agent to make decisions based on the location of other agents on the field. The framework does not have any beans to facilitate explicit inter-

agent communication either. A behavior bean that sends “say” messages to the server would be useful. Finally, there is always scope for improving the low-level abilities of the agents. We can do with an agent that performs a tighter dribbling, computes the kick power better, and uses “turn_neck” instead of “turn” wherever possible. Users would be able to design significantly better players if these beans were available.

Bibliography

1. Mikio Aoyama. *New Age of Software Development: How Component Based Software Engineering changes the way of Software Development*. In Proceedings of 1999 International Workshop on Component Based Software Engineering.

2. F. Bellifemine, A. Poggi, and G. Rimassa. *Developing multi agent systems with a fipa-compliant agent framework*. Software - Practice And Experience, 31(2):103--128, 2001.

3. F. Bellifemine, A. Poggi and G. Rimassi. *JADE: A FIPA-Compliant agent framework*, Proc. Practical Applications of Intelligent Agents and Multi-Agents, April 1999, pg 97-108.

4. T. Berners-Lee, J. Hendler, and O. Lasilla. *The Semantic Web*. Scientific American, 2001.

5. Biter: A RoboCup client. `http://jmvidal.cse.sc.edu/biter/`.

6. G. Booch. *Object-Oriented Design with Applications*. Benjamin/Cummings, Redwood City, CA, 1991.

7. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

8. R. A. Brooks. *Intelligence without representation*. Artificial Intelligence, vol. 47, pp. 139 -- 159, 1991.

9. A. W. Brown, Editor. *Component-Based Software Engineering*. IEEE Computer Society, 1996.

10. A. W. Brown and K. C. Wallnau. *The Current State of CBSE*. IEEE Software, vol. 15, 1998, pp. 37-46.

11. P. A. Buhler, and J. M. Vidal. *Biter: A Platform for the Teaching and Research of Multiagent Systems' Design using Robocup*. RoboCup 2001: Robot Soccer World Cup V.LNCS/LNAI Lecture Notes Volume 2377. Springer Verlag, Berlin Heidelberg New York (2002).

12. D. Camacho, R. Aler, C. Castro, J. M. Molina. *Performance Evaluation of Zeus, JADE and SkeletonAgent Frameworks*. IEEE nternational Conference on Systems, Man, and Cybernetics 2002 (SMC-2002), October 2002, Hammamet, Tunisia.

13. Enterprise JavaBeans Technology. `http://java.sun.com/products/ejb/`

14. M. Fayad and D. C. Schmidt. *Object-Oriented Application Frameworks*. Communications of the ACM, 40(10), 1997.

15. Steven P. Fonseca, Martin L. Griss, Reed Letsinger, *Agent Behavior Architectures, A MAS Framework Comparison*, Hewlett-Packard Labs, Technical Report, HPL-2001332, (short paper form at AAMAS 2002)

16. Foundation for Intelligent Physical Agents. `http://www.fipa.org/`

17. E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

18. M. R. Genesereth, and S. P. Ketchpel. *Software Agents.* Communications of the ACM, vol. 37, no. 7, 1994, pp.48-53.

19. M. Georgeff, B. Pell, M. Pollack, M. Tambe, and M. Wooldridge. *The belief-desire-intention model of agency*. Proceedings of Agents, Theories, Architectures and Languages (ATAL), 1999.

20. H. Goradia, and J. M. Vidal. *Building Blocks for Agent Design.* Paolo Giorgini, editor, Agent-Oriented Software Engineering, LNCS, Springer-Verlag, 2004. to appear.

21. J. Hendler, *Agent and the Semantic Web.* IEEE Intelligent Systems, Vol. 16, No. 2, 2001, pp. 30--37.

22. B. Horling. *A Reusable Component Architecture for Agent Construction*. University of Massachusetts/Amherst CMPSCI Technical Report 1998-49. October, 1998.

23. C. Iglesias, M. Garijo, and J. Gonzales. *A survey of agent-oriented methodologies*. A. S. Rao, J.P. Muller and M. P. Singh, editors, Intelligent Agents IV (ATAL98), LNAI. Springer-Verlag, 1999.

24. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

25. JADE. `http://sharon.cselt.it/projects/jade/`

26. JavaBeans: The Only Component Architecture for Java Technology. http://java.sun.com/products/javabeans/

27. N. R. Jennings. *On agent-based software engineering*. Artificial Intelligence, 117(2):277-296, 2000.

28. Kyo C. Kang, *Issues in Component-Based Software Engineering*. '99 International Workshop on Component-Based Software Engineering, 21st ICSE, Los Angeles, CA, May 1999.

29. .NET Framework Home. http://msdn.microsoft.com/netframework/

30. J. Ning. *A Component-Based Software Development Model.* Proceedings of 20th Computer Software and Applications Conference, Aug 1996, pp. 389-394.

31. I. Noda. *Soccer server: A simulator of robocup*. In Proceedings of AI symposium '95, pages 29--34. Japanese Society for Artificial Intelligence, December 1995.

32. I. Noda, H. Matsubara, K. Hiraki, and I. Frank. *Soccer server: a tool for research on multiagent systems*. In Applied Artificial Intelligence, volume 12, pages 233--250, 1998.

33. H. Nwana, D. Ndumu, L. Lee, and J. Collis. *ZEUS: A tool-kit for building distributed multi-agent systems*. Applied Artifical Intelligence Journal, 13(1):129--186, 1999.

34. OMG's CORBA Website. http://www.corba.org/

35. PRS-CL Architecture. http://www.ai.sri.com/~prs/prs-arch.html

36. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design* , Englewood Cliffs, NJ, Prentice Hall, 1991.

37. M. Shaw, and D. Garlan. *Software Architecture.* Prentice Hall, 1996.

38. Y. Shoham. *Agent-oriented programming*. Artificial Intelligence, 60(1):51--92, March 1993.

39. Soccer Server System. http://sserver.sourceforge.net/

40. Peter Stone. *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer.* The MIT Press, 2000.

41. C. Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison Wesley Longman, Reading, Mass., 1998.

42. J. M. Vidal, P. A. Buhler. *Teaching Multiagent Systems using RoboCup and Biter.* The Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, (4)2, 2002.

43. Jose M. Vidal, Paul A. Buhler, and Michael N. Huhns. *Inside an agent*. IEEE Internet Computing, 5(1), January-February 2001.

44. W3C Semantic Web. `http://www.w3.org/2001/sw/`

45. W3C Web Services. `http://www.w3.org/2002/ws/`

46. M. Wooldridge and P. Ciancarini. *Agent-Oriented Software Engineering: The State of the Art* Handbook of Software Engineering and Knowledge Engineering: World Scientific Publishing Co., 2001.

47. M. Wooldridge and N. R. Jennings. *Intelligent Agents: Theory and Practice*. Knowledge Engineering Review, 10(2), June 1995. Cambridge University Press, 1995.

48. M. Wooldridge and N. R. Jennings. *Pitfalls of agent-oriented development*. Proceedings of the Second International Conference on Autonomous Agents (Agents 98), pages 385--391, Minneapolis/St Paul, MN, May 1998.

Appendix A

SoccerBeans source code

Source code for the SoccerBeans package is available as a JAR file at the SoccerBeans homepage. It is located at the following URL:

http://jmvidal.cse.sc.edu/soccerbeans

The SoccerBeans package also includes bytecode and javadoc, along with a short document describing the various beans defined in the framework.

Appendix B

SoccerBeans tutorial

This chapter is supposed to serve as a tutorial for creating a simple player using SoccerBeans. You will learn how to create and run a simple player that can dribble the ball to goal using the SoccerBeans framework.

B.1 Install the Soccer Server

1. Refer to the RoboCup soccer simulator's home page for further information.

2. Students at USC can use the server installed on the SUN machines at **`/acct/f1/jmvidal/SUN/rcsoccersim-9.0.3`**. Run the shell script **`rcsoccersim`** from there.

B.2 Install and Setup the BDK

1. Refer to the JavaBeans Architecture: BDK Download home page for further details.

2. As mentioned in the website, there are some unresolved compatibility issues concerning J2SE v 1.4 and the BDK. So, we will have to use some older J2SE version, preferably J2SE v 1.3 (The software was tested for J2SE v 1.3.0 and J2SE v 1.3.1_04). Students at USC must modify the **`run.sh`** file at the location **`$(BDK_HOME)/beans/beanbox`** to

```
#!/bin/sh
export PATH
PATH=/usr/local/java1.3/bin:$PATH
export CLASSPATH
CLASSPATH=classes:../infobus.jar:../lib/methodtracer.jar
java sun.beanbox.BeanBoxFrame
```

   where, `$(BDK_HOME)` represents the BDK installation directory.

3. Copy the server configuration file, **`.rcssserver-server.conf`**, from the soccer server's home directory to the **`$(BDK_HOME)/beans/beanbox`** directory. The SoccerBeans package includes a sample server configuration file. Students at USC must use the server configuration file defined in FP and rename it to **`.rcssserver-server.conf`**

4. Save the SoccerBeans package in the **`$(BDK_HOME)/beans/jars/`** directory. To avoid inconvenience, move all the other contents of the directory to some new location.

B.3 Create a Soccer Player

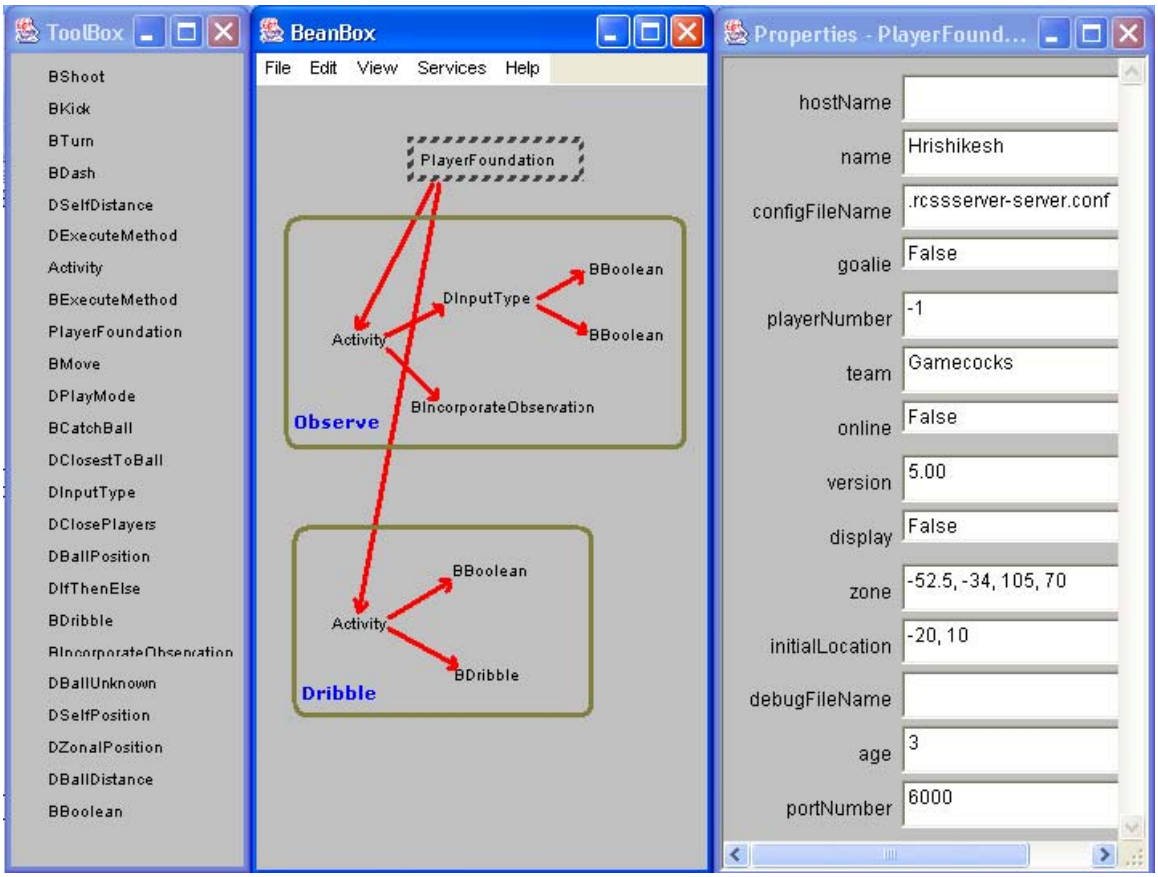


Figure A.1. A dribbling player

1. Start the BDK application by executing **`./run.sh`** from the **`$(BDK_HOME)/beans/beanbox`** directory. Four windows - BeanBox, Toolbox, Properties and Method Tracer, will appear. Also, the Toolbox will have all the beans from the SoccerBeans package.

2. Click on the **`PlayerFoundation`** bean from the Toolbox window and then click somewhere in the top section of the BeanBox window. We now have a soccer player that can play the game, if we tell it what to do.

3. Now we will add an activity that tells the player to incorporate the new information that it receives from the server at runtime into it's world model.

   - Drop the **`Activity`** bean in the BeanBox, as shown in the figure A.1. Change it's *name* property to `Observe`.
   - Drop the other beans for the `Observe` activity, namely the **`DInputType`** bean, the two **`BBoolean`** beans and the **`BIncorporateObservation`** bean on the BeanBox.
   - Select the **`DInputType`** bean in the BeanBox. It's *inputType* property must be set to `SensorInput`. Now select the `Observe` activity bean, and from the `Edit` menu, go to `Events > canHandle Decision > decide`. A red line will appear connecting the mouse pointer to the activity bean. Then click on the `DInputType` bean, and in the window that appears, select `decide` and `OK` it. By this action, we say that the `Observe` activity can be handled or executed by the player when a particular condition on the type of the input received from the soccer server is satisfied. The `canHandle Decision > decide` event was selected for the source bean (Activity) because the listener bean (DInputType) was a decision bean.

- Select the first **`BBoolean`** bean. By default, the bean's properties are set such that it sets the *canHandle* property of the concerned activity to `true`. Now select the **`DInputType`** bean, and go `Edit > Events > then behave > behave`. Select the first **`BBoolean`** bean and say `select` and `OK`. Here we say that if the received input type is a `SensorInput`, i.e. `see` message from the server (see the server manual for more information), then the concerned activity, i.e. `Observe`, can be handled. The `then behave > behave` event was selected for the source bean (DInputType) in this case, as the listener bean (BBoolean) was a behavior bean.
- Select the second **`BBoolean`** bean. Change it's *value* property to `false`. Then select the **`DInputType`** bean, go `Edit > Events > else behave > behave`, select the second **`BBoolean`** bean, and say `behave` and `OK`. Here we say that if the received input type is not a `SensorInput`, then the `Observe` activity cannot be executed by the player for the current simulation step.
- Observe that the **`BIncorporateObservation`** bean has no properties. Select the `Observe` activity bean, go `Edit > handle Behavior > behave`, select the **`BIncorporateObservation`** bean, and say `behave` and `OK`. Here we described how the player will execute the activity, should the player's scheduling mechanism select this activity for execution in the current simulation cycle. We say that the player will simply call the **`BIncorporateObservation`** bean to handle the current activity.
- Finally, we have to associate our `Observe` activity to the player. For this, select the **`PlayerFoundation`** bean. The **`PlayerFoundation`** bean can

generate an addActivity event, so go Edit > Events > activity > add activity, then select the Observe activity bean, and select it's addActivity method. Repeat the process for the **PlayerFoundation** bean's canHandle and handle events, associating those with the canHandle and handle methods of the activity bean. We have now successfully designed one activity for our player.

4. We will next add an activity that teaches the player to dribble the ball.

   - Drop the **Activity**, **BBoolean** and **BDribble** beans from the Toolbox onto the BeanBox, as shown in the adjacent diagram.
   - Select the **Activity** bean and modify it's *name* property to Dribble. Associate the activity's canHandle Behavior > behave event with the behave method of the **BBoolean** bean, and the handle behavior > behave event with the behave method of the **BDribble** bean. The default properties of the **BDribble** bean are set such that the player dribbles the ball to the opposite goal. Here we tell the player to execute this activity all the time.
   - Associate the Dribble activity with the player exactly as we associated the Observe activity.
   - There is one more thing that we need to take care of. We would like to give a higher priority to the Observe activity, as the player should always have the correct picture of it's environment. To do this, select the Observe activity, and

for it's *inhibits* property, say `Dribble`. So, we tell tha player to `Dribble` all the time, but whenever it can `Observe`, it must.

- Our player is now ready for action. Ensure that the soccer server is running, and select the **`PlayerFoundation`** bean in the BeanBox. The default values for the player's properties serve our current purpose, so we won't modify them. Simply set the *online* property to `true`. A player should appear on the soccer monitor. Click on **kick-off** and the player will do what we taught it.

Appendix C

Running the Sample Team

This chapter is supposed to serve as a tutorial for running the sample team provided in the SoccerBeans package, and described in chapter 5 of this report.

1. For reloading a JavaBeans application created in multiple sessions, we have to incrementally load every previous session in the BDK. So, say we create a player with one activity and save the session, and then load the player an hour later and create another activity for the player in this new session. Then, if we want to reload the player the next day, we will have to first load the file holding the first session's work and then the file holding the second session's work to open the complete player with two activities. An attempt to directly open the second session file will result in a **`ClassNotFoundException`** being thrown. The players in the sample team were developed in multiple sessions.

2. The sessions can be loaded by the **`File > Load`** option, and saved by the **`File > Save`** option in the BDK.

3. The **`sampleteam`** folder in the SoccerBeans package has 23 files for the 23 different sessions that the 11 players are created in. The **`observe`** session is the first session for all the players, and is always followed by the **`startplay`** session. The rest of the session files are named such that the alphabetically increasing order of individual player session files represents the actual session order for the player. So, to load **`player1`** completely, we will have to load the following files in the specified order - **`observe, startplay, player1a, player1b, player1c,`** and **`player1d.`** Note that some of the session files are shared by multiple players - **`player2a3a`** is the third session for both **`player2`** and **`player3`**, and **`player4c5c6c`** is the fifth session for **`player4, player5`** and **`player6`**.

P.S.: For convenience, copy the 23 sample team files in the **`$(BDK_HOME)/beans/beanbox`** folder.