Proceedings of the Fourth International Workshop on Agent-Oriented Software Engineering, pp. 17-30, 2003

Building Blocks for Agent Design

Hrishikesh J. Goradia

Swearingen Engineering Center
University of South Carolina
Columbia, SC 29208
goradia@engr.sc.edu

José M. Vidal

Swearingen Engineering Center
University of South Carolina
Columbia, SC 29208
vidal@sc.edu
http://jmvidal.ece.sc.edu

Abstract. We present our Component-Based Agent Framework, which enables a software engineer to design a set of agents by using a visual compo nent-based toolkit (Sun's BDK), and wiring together desired blocks of functionality. We instantiate this framework in the RoboCup domain by implementing the necessary components. The implementation also serves as a proof of the viability of our framework. Finally, we use this implementation to build sample agents. The proposed framework is a first step towards the merging of agent-based and component-based design tools.

1 Introduction

As the Semantic Web [6] and Web services become increasingly ubiquitous we can expect to see an increasing need for agents that can exploit these resources [9]. While some of the agents will be highly complex, we expect that most of them will be simple agents of limited abilities, short lifespan, and built in order to solve temporary problems. Software engineers that have to build these agents will, therefore, want methods that allow them to very quickly develop agents that have the desired capabilities. They will not want to spend time learning to use complex agent architectures in order to build an agent that might only be used a dozen times. They will instead prefer to use the tools that they have grown used to, such as visual component-based development systems.

In this paper, we present our Component-Based Agent Framework (CBAF) along with SoccerBeans—an example application of this framework for the RoboCup [13] domain. CBAF builds on our previous work on an agent architecture for RoboCup [5, 14] by integrating parts of the architecture with the Java Beans component model. The framework is our first step towards the merging of agent-based and component-based design methodologies and tools. Once our framework is instantiated for a particular domain, the resulting system allows the user to build agents using a visual component-based development program (BDK). In this way, the user does not need to think about the domain-independent aspects of building an agent and can instead focus solely on the domain-dependent aspects.

This material is based upon work supported by the NSF under Grant No. 0092593

2 The Component-Based Agent Framework Specifications

Despite the recent advances in the quality and the available features in current agent 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 CBAF specifications enable a software engineer to design multiagent systems by simply wiring together the desired blocks of functionality for each agent from a pool of available components, thereby circumventing the abovementioned problems. The frameworks based on the CBAF specifications are simple, generic, and flexible, imposing minimal restrictions and providing a plethora of options for designing agents that will participate in a multiagent system.

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 can be utilized by the user to develop sophisticated agent behaviors and associate the behaviors with individual agents. Most of the current agent frameworks implement a specific agent system, and a user of these systems is expected to download the provided software and build his 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. The developed agent system will be domain-specific, but by adding this restriction the learning curve for the user, before he can start building applications using the system, becomes virtually non-existent. Figure 1 compares CBAF with other current frameworks. CBAF does not provide any software libraries for agent applications, just a set of rules to be followed by the *designer* while developing agent systems that can be used by the *user* for creating agent applications. 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.

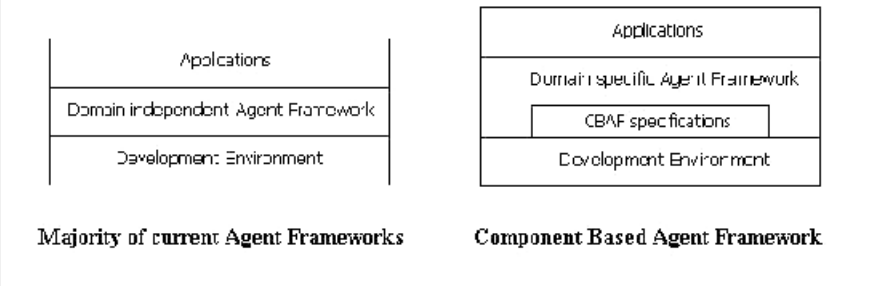

Fig. 1. Comparison of agent frameworks.

2.1 CBAF specifications

The CBAF specifications defining the necessary environment settings and the rules of agent system design to be followed by the designer and the user of the system are as follows:

- CBAF assumes a discrete, stochastic and episodic environment that can also be dynamic and partially observable with real-time requirements. Also, the set of actions that an agent can take must be bounded.
- CBAF is a component-oriented approach. The development environment selected for designing the agent system must support event-driven programming.
- The agent framework based on the CBAF architecture must be designed as a set of independent, self-contained, highly specialized components. A user must be able to combine the components in various forms to create different plans or activities for the agent. These activities will be associated to the individual agents.
- The agent 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 defied 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.

- *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 2 describes a sample CBAF architecture for the RoboCup domain.

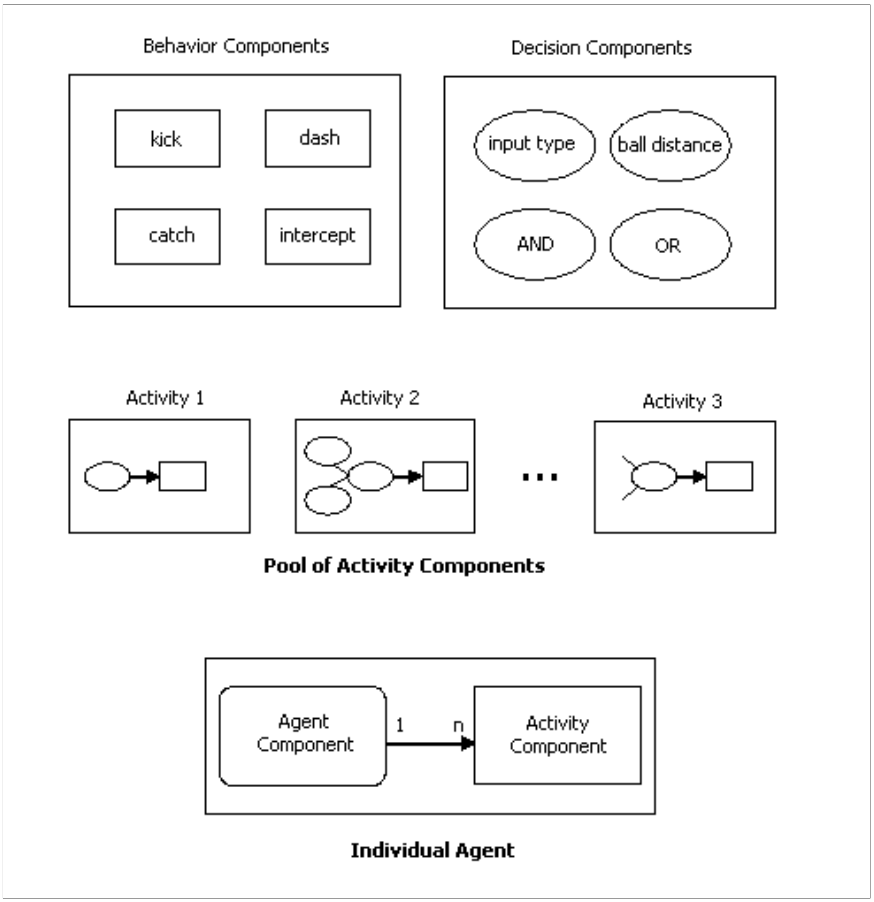

Fig. 2. The CBAF architecture. The sample behavior and decision components are shown for the RoboCup domain.

2.2 Challenges for CBAF design

Some of the issues that need to be addressed for designing the framework described above are:

- Which are the best behavior and decision components to provide to achieve the desired goals?
- 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 should the components be combined to create new components?
- How do we deploy the whole system with multiple agents?

2.3 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.

3 The SoccerBeans Framework

SoccerBeans is an implementation of the CBAF specifications for the RoboCup domain. The RoboCup domain is the world of simulated robotic soccer. The SoccerBeans agent system facilitates the creation of different soccer teams with minimal effort on the user's part. SoccerBeans is intended to be used as a pedagogical tool to instigate research in multiagent systems. RoboCup proves to be an excellent test-bed for study and research in multiagent systems' design as it presents a complex, distributed, real-time, noisy, collaborative, and adversarial environment for extensive research in agent based systems.

The RoboCup simulation system design is based on the client-server architecture. The soccer server provides a virtual field and simulates all movements of a ball and players, and controls a game according to rules. The multiagent system of soccer players forms the client side. Each client controls the movements of one player. Communication between the server and each client is done via UDP/IP sockets.

The SoccerBeans system consists of a pool of functional components developed as JavaBeans (referred to from hereon as just *beans*). The `PlayerFoundation` bean is a highly specialized agent component that represents an individual player in the multiagent system. The bean handles all the communication, knowledge representation, and activity scheduling aspects of the represented player. The `Activity` bean is the CBAF activity component that glues together the `PlayerFoundation` bean with the different player activities designed from the various decision and behavior beans provided in the system. Eight decision and behavior beans have been developed so far and work is in progress for developing additional beans to further extend the capabilities of the system. As will be shown later in this section, even with such a limited set of components, the CBAF implementation using JavaBeans can support substantially complex decision-making activities.

3.1 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. 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 dynamic information about the surrounding environment received from the server at every simulation step 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 actions a player can take. The activity scheduling mechanism of the `PlayerFoundation` bean supports both the BDI and the Subsumption architectures. The design of the `PlayerFoundation`

bean is very closely based on the Biter platform [5]. (Please refer to the paper on Biter for further information about the above mentioned classes in the `PlayerFoundation` bean.)

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 and after every goal can be set by the `initialLocation` property. `Age` defines the number of cycles for which the ghost of a dynamic object is preserved 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 connection between the server and the player.

As defined by the `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.

3.2 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.

As defined in the CBAF specifications, 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 `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.

3.3 Decision beans

As described in the CBAF specifications, 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 following decision beans have been developed for the SoccerBeans system to date:

3.3.1 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`.

3.3.2 DClosePlayers bean

The `DClosePlayers` bean can be used to create rules based on the number of other 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 exposes the properties `teamName, distance` and `number` for the above functions. The `DClosePlayers` bean executes the `playersInCone` method defined in the `RobocupUtilities` class to determine the result. The `decision` flag is set to `true` if the number of players counted is less than the value of the number property, else it is set to `false`.

3.3.3 DBallDistance bean

The `DBallDistance` bean can be used for classifications based on the distance of the ball from the player. The `DBallDistanceBeanInfo` class exposes the `DBallDistance`'s `distance` property, which allows a user to specify the threshold distance. The actual distance is determined from the player's world model. The `decision` flag is set to `true` if the determined distance is less than the value of the distance property, else it is set to `false`.

3.3.4 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. The `DIfThenElseBeanInfo` class provides an additional set of add and remove listener methods for the new chain.

3.4 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 following behavior beans have been developed for the SoccerBeans system to date:

3.4.1 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.

3.4.2 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 `BIncorporateObservation-BeanInfo` class is defined accordingly.

3.4.3 BDribbleBallToPoint bean

The `BDribbleBallToPoint` bean allows the player to dribble the ball to the specified point on the field. A user can specify the point at design time through the `final-Position` property. This feature can be useful for testing purposes, but has limited applicability at run time as the desired final position for the ball in the soccer game will almost always vary at different points in time. The user would rather like to execute some method for computing the desired final position for the ball at a given time. The `BDribbleBallToPoint` bean provides this option by presenting other fields to the user at design time. The `BDribbleBallToPointBeanInfo` class also exposes the `method`, `className`, `methodName` and `methodParams` properties of the bean. The `className`, `methodName` and `methodParams` properties can be used to specify the appropriate method with parameters that should be executed for computing the desired final position of the ball. The method is invoked at runtime using the Reflection API. The `method` flag is used to determine whether the ball's final position must be computed by executing the specified method or by considering the value of the `finalPosition` field. The bean executes the `dribbleBallToPoint` method defined in the `RobocupUtilities` class, passing the desired final position value as the parameter.

3.4.4 BShootBallToPoint bean

The `BShootBallToPoint` bean is similar to the `BDribbleBallToPoint` bean except that the latter dribbles the ball, while the former kicks it with maximum force. The `BShootBallToPointBeanInfo` class provides similar options to the user as the `BDribbleBallToPointBeanInfo` class. This bean executes the `shootBallTo-Point` method in the `RobocupUtilities` class.

4 Testing and Results

Using the developed components in SoccerBeans, we designed a basic soccer agent that was capable of dribbling a ball to its goal. The agent could also shoot the ball to a desired point in the soccer field if other players surrounded it. Figure 3 shows the agent design on the BDK.

The agent can perform three activities: `observe, dribble` and `shoot`. These are ordered from top to bottom in Figure 3. For every cycle in the agent's scheduling mechanism, if the player has received a new sensor input from the soccer server, then the BDI part of the mechanism selects the `observe` activity for execution. For action event inputs, the dribble and shoot activities are selected. The `dribble` activity is eligible for all action event inputs, while the `shoot` activity is eligible only for cases where other players are closing in on the agent. The subsumption part of the scheduling mechanism breaks the tie for such cases. The shoot activity always inhibits the dribble activity in the agent design.

The soccer game was played between two teams consisting of a single player, each designed as shown in the above figure. Successful experiments were conducted also for teams with a couple of players each, where the shoot activity was set to fire if any opponent approached the agent. Work is in progress for developing more components in SoccerBeans to enable the players to make non-trivial decisions for advanced communication and coordination.

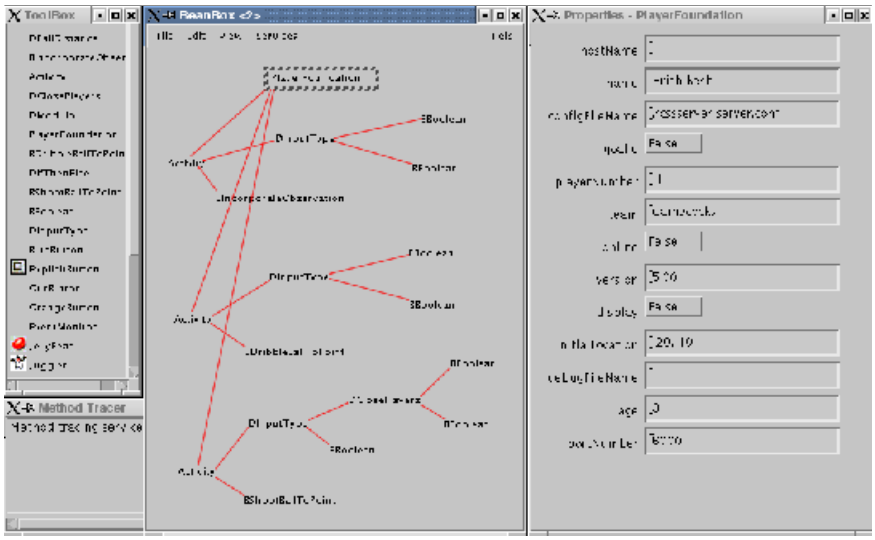

Fig. 3. Agent design using SoccerBeans. The red lines showing the connections between the components were added externally to the actual screenshot.

5 Related Work

Over the years, researchers have developed many agent frameworks for creating multiagent systems. Some of the frameworks like JADE [2] and ZEUS [12] are domain-independent, but they restrict their users to designing agents and agent systems in a specific way. A user needs to learn the architecture of these tools before being productive, which is not a trivial task considering the complexity of these frameworks. The main objective of our work is to enable the user to design agent systems using CBAF-compliant frameworks with minimal learning. SoccerBeans builds on our previous work on Biter, an agent architecture for RoboCup. Biter implements a client for the RoboCup simulator, providing the basic functionality to design RoboCup teams. Some of its features include a world model with absolute coordinates, a graphical debugging tool, a set of utility functions, and a generic agent architecture supporting both reactive (subsumption) [4] and practical reasoning (BDI) [8] responses for scheduling activities. SoccerBeans inherits all of the above features from Biter, and extends the architecture by integrating it with a component-based architecture – JavaBeans. The University of Massachusetts's JAF [10] project develops an agent framework using the JavaBeans technology. The JAF framework also provides components for designing disparate agents, but it is very restrictive in its methods for designing agents. In addition, the JAF system works only with the Multi Agent Survivability Simulator, a special test bed developed at the University of Massachusetts.

6 Conclusions and Future Work

We have introduced our CBAF specifications for developing agent frameworks, along with SoccerBeans—an implementation of CBAF for the RoboCup domain. The CBAF-compliant frameworks are unique in that there is virtually no learning curve for their 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 blocks of functionality from a pool of available components. Our system allows the user to quickly design and deploy his agents without worrying about the underlying architecture. CBAF seems ideal for research on applications such as trading-agent systems and resource allocation problems where there is a need for many agents with only slightly different functionality. We have demonstrated the ease and the usefulness of CBAF specifications with our SoccerBeans implementation for the simulated soccer application. We intend to complete the SoccerBeans framework in the future by developing additional components to add versatility to the available features for agent design. Finally, we view our system as a prototype for future agent-development environments that will merge the best techniques from agent-based software engineering and component-based design.

References

1. Biter: A robocup client.: http://jmvidal.cse.sc.edu/biter/.
2. Bellifemine, F., Poggi, A., and Rimassa, G.: Developing multi-agent systems with JADE. In: Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages, 2000.
3. Booch, G., Rumbaugh, J., and Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999.
4. Brookes, R.A.: Intelligence without representation. Artificial Intelligence, 47:139-159, 1991.
5. Buhler, P., Vidal, J.M.: Biter: A Platform for the Teaching and Research of Multi-agent Systems' Design using Robocup. In: RoboCup 2001: Robot Soccer World Cup V.LNCS/LNAI Lecture Notes Volume 2377. Springer Verlag, Berlin Heidelberg New York (2002).
6. Berners-Lee, T.,Hendler, J., and Lasilla, O.: The Semantic Web. Scientific American, 2001.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
8. Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M.: The Belief-Desire-Intention model of agency. In: Proceedings of Agents, Theories, Architectures, and Languages, 1999.
9. Hendler, J.: Agents and the Semantic Web. IEEE Intelligent Systems, (16)2, 2001.
10. Horling, B.: A Reusable Component Architecture for Agent Construction. In: University of Massachusetts/Amherst CMPSCI Technical Report 1998-49. October, 1998.
11. JavaBeans: The Only Component Architecture for Java Technology. http://java.sun.com/products/javabeans/
12. Nwana, H., Ndumu, D., Lee, L., and Collins, J.: ZEUS: A tool-kit for building distributed multi-agent systems. Applied Artificial Intelligence Journal, 13(1): 129-186, 1999.
13. Soccer Server System.: http://sserver.sourceforge.net/
14. Vidal, J.M., Buhler, P.: Teaching Multiagent Systems using RoboCup and Biter. The Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, (4)2, 2002.