A User Oriented System for Developing Behavior Based Agents

Paul Scerri, Silvia Coradeschi and Anders Törne

Department of Computer and Information Science
Linköping University, Sweden
Email: pausc@ida.liu.se, silco@ida.liu.se, ato@ida.liu.se

Abstract. Developing agents for simulation environments is usually the responsibility of computer experts. However, as domain experts have superior knowledge of the intended agent behavior, it is desirable to have domain experts directly specifying behavior. In this paper we describe a system which allows non-computer experts to specify the behavior of agents for the RoboCup domain. An agent designer is presented with a Graphical User Interface with which he can specify behaviors and activation conditions for behaviors in a layered behavior-based system. To support the testing and debugging process we are also developing interfaces that show, in real-time, the world from the agents perspective and the state of its reasoning process.

1 Introduction

Intelligent agents are used in a wide variety of simulation environments where they are expected to exhibit behavior similar to that of a human in the same situation. Examples of such environments include RoboCup[10], air combat simulations[14] and virtual theater[16].

Defining agents for simulation environments is a very active research area. The research has resulted in a large number of agent architectures being proposed. Many of the proposed architectures have accompanying languages for defining the behaviors, for example [1, 3, 5, 6, 11, 15, 16]. However many of these methods for specifying behavior are oriented towards a computer experts way of thinking, rather than to a domain experts, i.e. they use logic or other kinds of formalisms.

The quality of the behavior exhibited by an agent is closely related to the quality and quantity of the knowledge held by the agent. As domain, rather than computer, experts are likely to have superior knowledge of intended agent behavior, it seems to be advantageous to develop methods whereby domain experts can directly specify the behavior of the agents. It may be the case, especially in simulation environments, that parts of the behavior of an agent change often over the life of a system, in which case it is even more desirable to enpower domain experts to define and update behavior.

When developing agents with complex behavior for real-time complex environments it is often hard to debug and tune the behavior in order to achieve

M. Asada and H. Kitano (Eds.): RoboCup-98, LNAI 1604, pp. 173-186, 1999.

the desired result[8, 13]. When incomplete and uncertain information are added to the cocktail, as occurs in many domains including RoboCup, determining the reason for unwanted behavior can become extremely difficult.

The goal of allowing non-computer experts to specify complex behavior of simulated agents quickly and easily is a lofty one. In this paper we present the design of a system that addresses three aspects of the problem as it relates to RoboCup, namely: vertical rather than horizontal decomposition of behaviors; specification of conditions and behaviors in a high level abstract natural language manner; and a short and simple design-specify-debug cycle. The underlying ideas are not new, we have mainly pieced together existing ideas simplifying or adapting where necessary in order to create an environment that is simple for non-computer experts. Where possible we have tried to make the way a user specifies behavior as close as possible to the way human coaches would explain behavior to their players.

The system we are developing allows a user to specify the behaviors for a layered behavior based system via a Graphical User Interface(GUI). Activation conditions for behaviors are in the form of abstract natural-language like statements, which we refer to as predicates. The runtime system maps the natural language statements to fuzzy predicates.

The behavior specification interface presents the user with a window where they can define behaviors for each layer of a behavior based decision making system. The user can specify an appropriate activation predicate and a list of lower level behaviors, with associated activation information, that implements the functionality of the behavior. At runtime the behavior specification is used as input to a layered behavior based controller which uses the specification, along with world information abstracted from the incoming percepts, to turn low level control routine skills on and off as required. A GUI is provided to show in real time the way an agent perceives the field.

The choice of a behavior based architecture as the underlying architecture seems to be a natural choice as the structure of the architecture seems to correspond well to the way a human coach would naturally explain behavior. [1] A behavior-based architecture uses vertical decomposition of overall behavior, i.e. into defend and attack, rather than horizontal decomposition, i.e. into navigate and plan. For example a coach is likely to divide his discussion of team tactics into discussions on attacking and defending - this would directly correspond to attacking and defending behaviors in a behavior based system.

We use natural language statements which map to fuzzy predicates as a way of allowing an agent designer to specify conditions for behavior in his/her own language. The use of abstract natural language statements about the world as behavior activation mechanisms attempts to mimic the way a human might describe the reason for doing something. For example, a coach may tell a player

[1] This may or may not correspond to the way human decisions are actually made. However, the relevant issue is trying capture the experts explanation of the behavior rather than copying the decision making process.

to call for the ball when he is in `free space`. The idea of *free space* is a vague one, hence the natural choice of fuzzy predicates as an underlying implementation.

An artifact of behavior based systems is that they are difficult to predict before testing and, usually, also difficult to explain when observed. The process is further complicated in environments where incoming information is incomplete and uncertain. Consequently behavior based systems must go through a design-test cycle many times. To assist in the development process we have developed a real-time GUI interface that shows the world as the agent sees it and we have developed an interface that shows the state of the agents reasoning process i.e. which behaviors are executing and the activation level of non-executing behaviors.

Developing systems which allow non-computer experts to define agents is an active research area. Different approaches are often successful at simplifying the specification of behavior for a particular domain. Strippgen has developed a system for defining and testing behavior-based agents called INSIGHT [13]. It is claimed that a graphical representation of the internal state of an agent coupled with a visualization environment aids in testing and debugging agents. The INSIGHT system also includes an interface for incremental development of behaviors. Firby's RAP's system uses Reactive Action Packages to turn low level control routines on and off[6]. Our system is similar in that it provides an abstract method for deciding which skills to turn on and off, however we believe that RAP's is more suited to relatively static domains where the activities to be performed involve sequential tasks, with possibly a number of different available methods for achieving the goal, whereas our system is more suited to very dynamic domains. Moreover the usability aspect is not especially considered in RAPS. Harel has developed an extension of state machines called Statecharts[7] which is a powerful formalism for the representation of system behavior. However Statecharts are usually only used for the specification of complex physical system behavior. HCSM [4] is a framework for behavior and scenario control which uses similar underlying ideas to Statecharts. Like Statecharts HCSM is a very powerful way of representing behavior however it is not designed for easy specification by non-expert users. At the other end of the ease-of-use spectrum is KidSim [12] which allows specification of only simple behavior. As the name suggests, KidSim allows children to specify the behavior of agents in a dynamic environment via a purely graphical interface. An alternative way of specifying the behavior of agents is to have agents follow scripts like actors in a theater [16]. In [3] a system is presented that also has the aim of making behavior specification easier, however in that system no GUI is present and a different decision making mechanism is used.

Currently we are applying our system to the specification of RoboCup agents. However, we intend in the future to adapt it for specifying agents for air-combat and rescue simulations.

2 How a User Perceives the System

We are developing a system which allows users who are not necessarily programmers to specify the complex behavior of an agent for a complex domain. An agent designer can define complex agents by defining layers of behaviors, specifying activation conditions and testing agents all without having to write or compile any code.

An agent definition is in the form of an arbitrary number of layers of behaviors where behaviors on higher levels are more complex and abstract. At any time a single behavior on each level is executing. In higher levels the executing behavior implements its functionality by specifying the lower level, less abstract behaviors that should be considered for execution. The bottom level behaviors execute by sending commands to an interface which in turn turns on or off low level *skills*. The selection of the behavior to execute is determined by finding the behavior with the highest *activation* at that point in time. The activation level of a behavior is a time dependent function that depends on the truth of the fuzzy predicate[2] underlying the natural language statement and user specified activation parameters associated with the behavior.

The development process consists of developing behaviors for lower levels, testing the partially specified agent, then using previously developed behaviors to specify behaviors for higher layers. Lower layers, even incomplete lower layers, can be fully tested and debugged before higher level behaviors are specified.

A behavior is defined by specifying a name, the level of the system the behavior is at, a predicate for the behavior and a list of, possibly parameterized, lower level behaviors with associated activation parameters that together implement the functionality of the behavior. This is all done via a graphical user interface. At runtime the user can observe, via another GUI, the interactions between behaviors.

Another window provides real time feedback on exactly how the player perceives the world. This enables an agent developer to better understand the information upon which the agents reasoning is being done and therefore design a better set of behaviors. The GUI is relatively decoupled from the rest of the system and is in our intention to make it publicly available for other developers to use.

2.1 Creating Behaviors

When the users first starts up the specification system a main window opens up. This window gives the user the opportunity to open previously saved agent behavior specifications or to start a new behavior specification. The main window shows the behaviors that have been created previously for this agent. These behaviors can be used to implement the functionality of new higher level behaviors.

[2] We use this term very loosely to indicate a function that returns a value between `true` and `false` to indicate the perceived truth of some statement.

The user can choose to edit an existing behavior or create a new behavior. Clicking on either the *New Behavior* or the *Edit Behavior* button pops up a second window (see Figure 1). Here the user specifies information about the nature of the behavior.

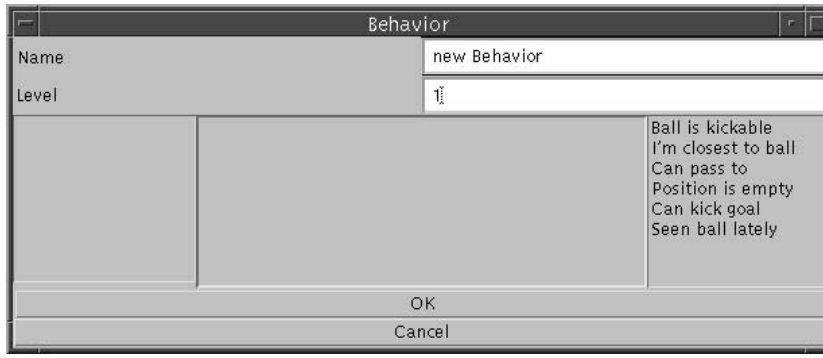

Fig. 1. The Behavior Specification Window. This is how the window appears when the *New Behavior* button is pressed. The list on the right shows the predicates that can be used by the behavior. The, initially empty, list on the left shows names of behaviors that can be used to implement the functionality of the behavior. The middle of the window will show selected behaviors and their associated activation parameters.

Once the user has finished specifying the behavior he can click OK and the behaviors name appears in the main window. The specification can then be saved and tested in the RoboCup simulator. Alternatively the user may directly begin work on other behaviors or a new higher level behavior which uses the previously defined behavior. There is no compilation required when the behaviors for an agent are changed.[3] However, real-time performance is still achieved as the control system that executes the behaviors is compiled and acts sufficiently quickly.

The way a user perceives different elements of a behavior specification does not necessary correspond to the actual underlying implementation. The intention is that the users are presented with an interface that allows them to express their ideas as naturally as possible and the underlying system takes the specification and uses it to make decisions.

The information required to fully specify a behavior is the following:

- Name
- Level
- Predicate
- List of Behaviors

[3] It is anticipated that eventually the behaviors will be able to be changed on line. However, at present the agent must be restarted when behaviors are modified.

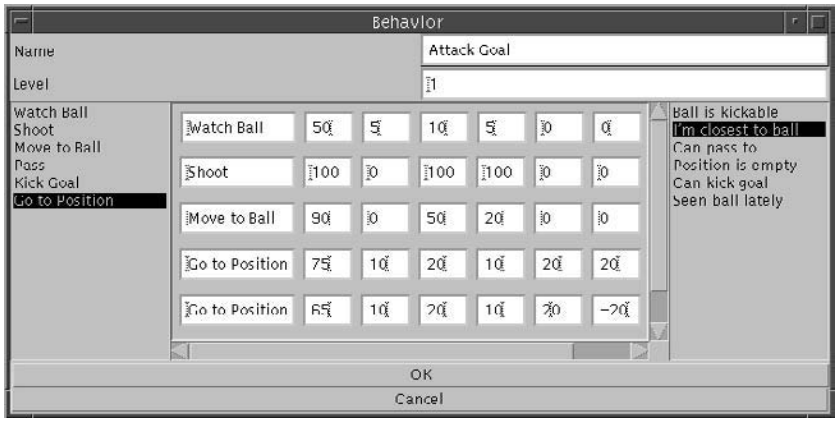

Fig. 2. The Behavior Specification Window with a behavior almost fully defined. In the middle is the list of behaviors that implements the functionality of `Attack Goal`. The numbers associated with each behavior are activation parameters (described below). In the list there are two instantiations of `Go to position`. The first `Go to position` is preferred as its maximum applicability, 75, is higher.

Each of these elements is discussed seperately below.

Name

Each behavior has a unique *name*. A designer specifying an agent can use names that abstractly represent the idea the behavior is trying to capture. Examples of behavior names for the RoboCup domain are `kick goal` - at a relatively low level of abstraction, `attack down left` - at a higher level of abstraction or `our free kick` - at an even higher level of abstraction. The names of behaviors on low levels are then used to specify behaviors on the next level up. The idea is that an agent designer can implement a behavior such as `our free kick` in terms of behaviors like `attack left` and `kick to open space`. The underlying system uses the behavior name as an identifier for the behavior.

Level

The *level* specifies which layer of the behavior based structure the behavior is to be used at. Different agents may have different numbers of levels depending on the design. The agent designer uses the level number to capture the intuitive idea that behaviors occur at different levels of abstraction - `moving to the ball` is at a low level of abstraction whereas `attacking` is at a high level of abstraction. Perhaps slightly less intuitive is the idea that very abstract behaviors are a result of interactions between a number of slightly less abstract behaviors. [4]

Predicate

To the agent designer a *predicate* is a statement about the world for which the

[4] This is an underlying concept in behavior based systems that has yet to be conclusively shown to be correct. However, as the underlying agent architecture for this system is a behavior based one, it is necessary that the agent designer uses this idea.

level of truth changes as the world changes. These statements occur at different levels of abstraction. Some example predicates for the RoboCup domain are `ball close enough to kick` - at a low level of abstraction, `good position to shoot at goal`, and `attacking position` - at a higher level of abstraction. The activation level of the behavior increases as the truth of the predicate statement increases. To the underlying system a predicate is merely the name of an abstraction of information received from the server. It is implemented as a function that maps data from the environment to a fuzzy truth value according to a programmer definition. An example of a mapping is a predicate `close to ball` which is implemented as a polynomial function of the last seen distance to the ball.

List of Behaviors

The *list of behaviors* is a list of behaviors less abstract than the one being defined that together implement the functionality of the behavior. Effectively the list of behaviors forms a hierarchical decomposition of the behaviors functionality. The behaviors in the list should interact in such a way that the intended complex behavior emerges. The process of choosing the behaviors and activation conditions is a difficult one. The short design-test cycle and the interfaces to aid analysis of the interactions can make the process of choosing appropriate behaviors simpler.

Each of the behaviors in the list may have some associated parameters which determine, along with the predicate that was specified when the less abstract behavior was created, the activation characteristics of the behavior. In order to influence the activation of each of the behaviors the user specifies four values: *Maximum Activation*, *Minimum Activation*, *Activation Increment*, and *Activation Decrement.*

To a user Maximum Activation is the highest activation a behavior can have. Intuitively, when more than one behavior are applicable the applicable behavior with the highest Maximum Activation will be executed. This allows representation of priorities between behaviors. To the system Maximum Activation is a hard limit above which the controller does not allow the runtime activation level of the behavior above.

To the user Minimum Activation is the lowest activation a behavior can have. Intuitively when no behaviors are applicable the behavior with highest Minimum Activation is executed. To the system Minimum Activation is a hard limit below which the controller does not let the runtime activation level of the behavior below.

To a user Activation Increment is the rate at which the activation level of the behavior increases when its predicate is true.[5] The Activation Decrement is the rate at which the activation of the behavior decays over time. These two values are closely related. High values for both the Decrement and Increment create a very reactive behavior, i.e. it quickly becomes the executing behavior when its predicate statement is true and quickly goes off again when the statement becomes false. Relatively low values for the Increment and Decrement result in

[5] As the predicate is actually a fuzzy predicate the activation increase is actually a function of the "truth" of the predicate.

a behavior that is not activated easily but temporarily stays active even after its predicate has become false. In the RoboCup domain a behavior such as `Kick Goal` may have high Increment and Decrement values, i.e quick reaction, so that when the predicate statement `ball close enough to kick` becomes true the behavior immediately starts executing and quickly stops executing when the predicate becomes false, i.e the ball is not close enough to kick. Behaviors such as `Attack` may have low values for Increment and Decrement so that the player does not start executing the `Attack` behavior until the appropriate predicate, possibly something like `We are in an attacking position`, has been consistently true for some time. However, it maintains the `Attack` behavior even if the predicate becomes false for a short time - perhaps due to temporarily incorrect information. It was the authors experience with previous behavior based systems for RoboCup that much instability is caused by temporarily incorrect information mainly due to incomplete and uncertain incoming information.

The use of a list of behaviors and corresponding activation parameters allows multiple uses of the same lower level behavior in a single higher level behavior specification. For example, an attack behavior may use two instances of the lower level behavior `move to position` (having predicate `Position is empty`) with different activation parameters and position to move to. The result may be that the agent "prefers" to move to one position over another.

2.2 Debugging

Behavior based agents interact very closely with their environment. Interactions between the world and relatively simple behaviors combine in complex ways to produce complex observed overall behavior[2]. Although the resulting behavior may exhibit desirable properties the complex interactions that occur make behavior based systems extremely difficult to analyze and predict especially when they exist in dynamic, uncertain domains[9]. It can often even be difficult to determine the reasons for unwanted behavior simply by observing the overall, complex behavior of the agent. Therefore an important part of any system for creating behavior based agents is a mechanism for allowing the user to quickly test and debug agents. To this end we have developed a graphical interface which shows in real-time the world as the agent see it. We have also developed an interface which graphically shows the reasoning of the agent, i.e. the currently selected behaviors at each level and the activation levels of all behaviors.

The world information interface draws the soccer ground as the agent sees it, displaying information such as the agents calculated position, the calculated position of the ball, the position and team of other players and the status of the game. This interface is intended to make it easier for developers to determine the causes for unwanted behavior in an agent.

The designers can make more informed behavior designs when they have a better understanding of the information the agent has available.[6] For example,

[6] During the overall system development the interface has also proved useful in determining errors in the way the agent processes information.

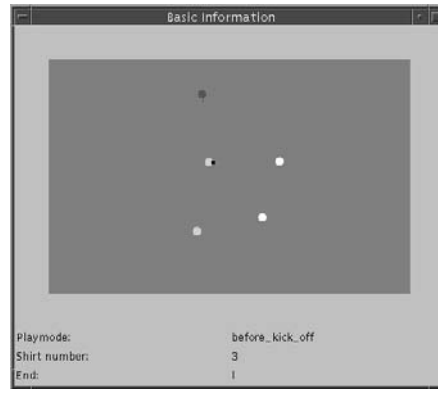

Fig. 3. The GUI showing how the agent perceives the world. The dark colored circle near the top of the screen is the agent whose world view is shown. Other circles represent the players in view. In the middle are team mates and on the right are opponents. Player of unknown team are shown in a different color (all players are known on the above diagram). Notice that the player that appears directly next to the agent of interest in the SoccerMonitor (See Figure 4) window does not appear in the agents perception of the world.

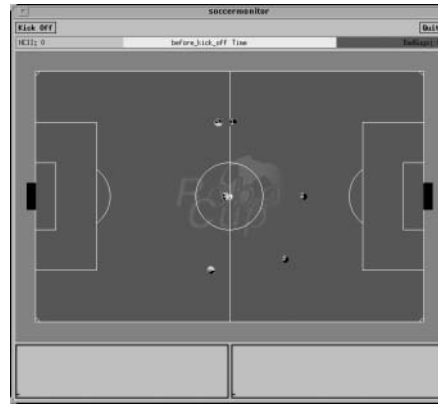

Fig. 4. The RoboCup Simulator showing the actual state of the world. Notice the two players close together near the top of the field. The player on the right does not appear in the player on the lefts view of the world - see Figure 3.

in Figure 3 the darker player near the top of the window perceives that he is in an empty space on the field although it is not, as can be seen from the RoboCup Soccermonitor (see Figure 4). The teammate with the ball can see that the player is not alone. This may possibly indicate to a designer that it is better for a player with the ball to look around for someone to pass to rather than relying on team mates to communicate that they would like to get the ball.

Also developed is an interface that displays in real-time the currently executing behavior on each level and the activation levels of all available behaviors (see Figure 5). This interface will allow designers to quickly determine the particular interactions that are resulting in undesirable behavior.

Fig. 5. A snapshot of the behavior activation window for an attacking player. Horizonatal lines represent the relative activation of the behaviors. Behaviors near the top of the window are higher level behaviors.

3 Underlying Agent Architecture

Many agent architectures have been developed, each with properties that make them suitable for some type of domain or some type of activity. For this system we use a layered behavior oriented architecture. The behavior activation mechanisms and behavior specifications are designed to allow the agents behavior to be specified without programming. Furthermore the entire agent system architecture is designed in such a way that it can accommodate a behavior based

decision making system that uses abstract predicates and acts by turning skills on and off.

The system is divided into seven sub-systems (see figure 6):

- *Information Processing:* The information processing sub-system is responsible for receiving incoming perceptual information and creating and maintaining an accurate view of the world for the agent.
- *Predicates:* The Predicate sub-system forms the interface between Information Processing sub-system and Behavior Based Decision Making subsystems and consists of a number of different fuzzy predicate objects. Predicates abstract away the details of the incoming information so that behaviors, and therefore agent designers, can use high level information for decision making.Rather than being precisely `true` or `false` predicates have a value that ranges between `true` and `false`.
- *Skills:* The Skills sub-system consists of a number of low level control routines for achieving particular simple tasks.The skills can have a very narrow scope such as moving towards a ball that has been kicked out of play.
- *Behavior Based Decision Making:* The Behavior based Decision Making system is responsible for the decision making of the agent. When the agent is started up an agent behavior description is loaded. At each layer of the system there is a controller which continually executes the following loop:
  - *Check if the layer above has specified a new set of behaviors. If so remove the old set of behaviors and get the new set.*
  - *Increase the* `activation level` *of all currently available behaviors by the value of the behaviors predicate times the* Activation Increment *value for the behavior.*
  - *Decrease the* `activation level` *of all currently available behaviors by the* `Activation Decrement` *value for the behavior.*
  - *If any behaviors* `activation level` *has gone above its* `Maximum Activation` *or below its* `Minimum Activation` *adjust the* `activation level` *so it is back within the legal range.*
  - *Find the behavior with the highest* `activation level` *and send the behavior list for this behavior to the next layer down (or in the case of the bottom level send a command to the interface).*
  - *Sleep until next cycle.*
- *Interface:*Interfaces between symbolic decision making systems and continuous control routines are an active area of research, e.g. [6]. We have implemented a very simple interface that may need to be extended in the future. The Interface receives strings representing simple commands from the decision making sub-system and reacts by turning on an appropriate skill in the Skills sub-system.
- *Debugging:* The Debugging sub-system acts as an observer of the rest of the agent architecture. The debugging sub-system works by periodically checking predefined information in the agent and representing the information graphically.

- *Server Interface:* The Server Interface is the sub-system responsible for communicating with the Soccer Server. The Server Interface sends incoming percepts to the Information Processing sub-system. Commands from the skills come to Server Interface to be sent to the Soccer Server.

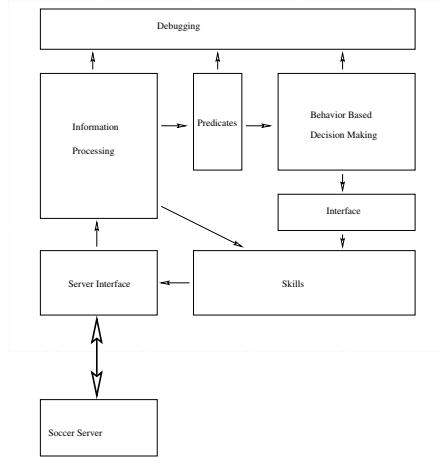

Fig. 6. An abstract view of the system architecture. The architecture of the agent is shown inside the dotted bow. Each box inside the agent represents a separate subsystem of the architecture. Arrows between sub-systems represent information flows.

The system was implemented in Java. Object Oriented techniques have been used in a way that allows new Skills and new Predicates to be quickly added by a programmer without changes being required to other parts of the system. For example the creation of a new predicate simply requires creating a subtype of an abstract Predicate class.

4 Evaluation

At the time of RoboCup98 in Paris the system was not sufficiently complete to allow evaluation with end users. However the authors used the GUI to specify a team which reached the quarter finals of the World Cup. During the development of the team and its subsequent use an evaluation based on observations was made. During the evaluation there were two main areas which were focused on, namely: the overall behavior of a finished agent; and the development process.

The overall observed behavior of the agent was reasonable. The agents consisted of around 40-50 different behaviors arranged into five or six levels. The agents were smoothly able to handle a range of different siutations at different levels of abstraction, for example agents exhibited different behavior in reponse

to almost all referee calls. A plesantly surprising aspect of the behavior of the agents was the way they handled uncertainty in information. During testing bugs in the timing of updates on the agents memory were noticed. However with some tuning of the activation increment and activation decrement parameters the bugs made little difference to the players behavior. A considerable weakness in the behavior was the inability of agent to combine output of active behaviors. The winner-take-all arbitration strategy of the layer controller means that only one behavior can act at any time. Although careful tuning of the specification avoided oscillations between behaviors, behaviors that should have been concurrently acting, most notably *move to position* and *avoid obstacle*, required extra programming at the skill level.

The development system, although promising, had a number considerable weaknesses. The GUI made it relatively simple and fast to make farily considerable changes to the behavior of an agent. The major problem with the interface was the need for an expert programmer to define predicates. Early in development almost every new behavior required low level coding of a new predicate. As the list of predicates became longer the need for the creation of new predicates became rarer but managing the list became more difficult. The requirement that each behavior be given a fixed level turned out to be rather inconvienient often requiring that *dummy* behaviors were created so that lower level behaviors could be used by higher layers.

The testing and debugging interfaces proved valuable when debugging a single agent however they were inadequate for debugging teams of agents. The interface showing what the player sees was mainly used for debugging information processing aspects of the agents rather than for debugging behavior of agents. The interface showing the state of the agents reasoning proved extremely valuable for determining the reason for incorrect behavior of an agent. The debugging interfaces turned out to be inadequate when trying to determine reasons for undesirable behavior in teams or team situations. The problem seemed to stem from the fact that it was impossible for a user to monitor the reasoning of 11 agents in real-time. Some system for focusing a users attention on important aspects of the reasoning process, or more simply record and playback facilities would be required to make the interfaces useful for multiagent debugging.

5 Conclusion

In this paper we have described a system that allows non-computer experts to specify the behavior of agents for the RoboCup domain. We also describe an interface that shows in real-time the world as the agent sees it and an interface that shows the state of the agents reasoning process. Future research will look at user acceptance of the system and work towards making the interface more intuitive to designers. Agents developed with this system competed at RoboCup'98 reaching the quarter finals.

Acknowledgments

Paul Scerri has been supported by the NUTEK project "Specification of agents for interactive simulation of complex environments". Silvia Coradeschi has been supported by the Wallenberg Foundation project "Information Technology for Autonomous Aircraft".

References

1. Bruce Blumberg and Tinsley Galyean. Multi-level control of autonomous animated creatures for real-time virtual environments. In *Siggraph '95 Proceedings*, 1995.
2. Rodney Brooks. Intelligence without reason. In *Proceedings 12th International Joint Conference on AI*, pages 569–595, Sydney, Australia, 1991.
3. Silvia Coradeschi and Lars Karlsson. *RoboCup-97: The First Robot World Cup Soccer Games and Conferences*, chapter A Role-Based Decision-Mechanism for Teams of Reactive and Coordinating Agents. Springer Verlag Lecture Notes in Artificial Intelligence, Nagoya, Japan, 1998.
4. James Cremer, Joseph Kearney, and Yiannis Papelis. HCSM: A framework for behavior and scenario control in virtual environments. *ACM Transactions on Modeling and Computer Simulation*, 1995.
5. Kieth Decker, Anandeep Pannu, Katia Sycara, and Mike Williamson. Designing behaviors for information agents. In *Autonomous Agents '97 Online Proceedings*, 1997.
6. James Firby. Task networks for controlling continuous processes. In *Proceedings of the Second International Conference on AI Planning Systems*, June 1994.
7. D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program*, 8:231–274, 1987.
8. Maja Mataric. Behavior-based systems: Main properties and implications. In *IEEE International Conference on Robotics and Automation, Workshop on Architectures for*, pages 46–54, Nice, France, May 1992.
9. Maja Mataric. *Interaction and Intelligent Behavior*. PhD thesis, Massachusetts Institute of Technology, 1994.
10. Itsuki Noda. Soccer server: A simulator of RoboCup. In *Proceedings of AI Symposium'95*, Japanese Society for Artificial Intelligence, December 1995.
11. Itsuki Noda. Agent programming in Gaea. In *RoboCup '97 Proceedings*, 1997.
12. David Smith, Allen Cypher, Jim Spohrer, Apple Labs, and Apple Computer. *Software Agents*, chapter KidSim: Programming Agents without a Programming Language. AAAI Press/The MIT Press, 1997.
13. Simone Strippgen. Insight: A virtual laboratory for looking into behavior-based autonomous agents. In *Autonomous Agents '97 Online Proceedings*, 1997.
14. Milind Tambe, W. Lewis Johnson, Randolph Jones, Frank Koss, John Laird, Paul Rosenbloom, and Karl Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 16(1), Spring 1995.
15. Sarah Thomas. *PLACA, An Agent Oriented Programming Language*. PhD thesis, Dept. Computer Science, Standford University, 1993.
16. Peter Wavish and David Connah. Virtual actors that can perform scripts and improvise roles. In *Autonomous Agents '97 Online Proceedings*, 1997.