Behavior Programming Language and Automated Code Generation for Agent Behavior Control

Thuc Vu, Manuela Veloso
tdv@andrew.cmu.edu, mmv@cs.cmu.edu
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA, 15213, USA

Abstract

Behavior-based agents are becoming increasingly used across a variety of platforms. The common approach to building such agents involves implementing the behavior synchronization and management algorithms directly in the agent's programming environment. This process makes it hard, if not impossible, to share common components of a behavior architecture across different agent implementations. This lack of reuse also makes it cumbersome to experiment with different behavior architectures as it forces users to manipulate native code directly, e.g. C++ or Java. In this paper, we provide a high-level behavior-centric programming language and an automated code generation system which together overcome these issues and facilitate the process of implementing and experimenting with different behavior architectures. The language is specifically designed to allow clear and precise descriptions of a behavior hierarchy, and can be automatically translated by our generator into C++ code. Once compiled, this C++ code yields an executable that directs the execution of behaviors in the agent's sense-plan-act cycle. We have tested this process with different platforms, including both software and robot agents, with various behavior architectures. We experienced the advantages of defining an agent by directly reasoning at the behavior architecture level followed by the automatic native code generation.

1. Introduction

There have been many research projects on different agent behavior architectures. Some behavior architectures, however, are applicable across domains and can be used to solve a variety of different control problems. However, reapplication of the same architecture in a different domain usually requires reimplementation of the architecture framework such as its behavior management code. It also requires extensive knowledge and manipulation of the native code, which is time consuming and error prone.

We conjecture that it will be beneficial to separate the process of designing agent behavior architecture for high-level behaviors and incorporating the architecture to a given agent platform with implemented platform-dependent atomic behaviors. A designer, once specialized in one agent architecture, can apply the architecture in existing agent platforms with minimum additional effort. Towards that goal, we provide an intuitive and flexible high-level behavior-based language (HLBL) that can be used to describe easily the architecture for an agent to facilitate the designing process, and a code generation system (B2C) to automate the integrating process. To our best knowledge, this work contributes the first platform-independent behavior-based programming language combined with an automated translation to native code.

Using the given high-level language, the designer can build an agent in the form of an augmented behavior hierarchy. This hierarchy will be specified in the behavior description file, which will indicate how the execution of behaviors should unfold at runtime. The given behaviors can be combined in different ways by the structure of the behavior hierarchy, leading to a variety of patterns of behaviors executed by the agent. This system allows changes in the behavior hierarchy to be made at an intuitive level without the need to change native code. Furthermore, when applying the architecture to a different domain, the designer can usually reuse part of the behavior description file. Thus the framework can significantly facilitate the process of developing and experimenting an agent behavior architecture.

Given the behavior hierarchy description file as input, the code generation system will generate the correspondent C++ code. After being translated, the behavior hierarchy is output as a ".cpp" file where each behavior in the hierarchy is a function in the .cpp file. The root of the hierarchy acts as the entry point to the agent control routines. The user then compiles and links this file with the rest of the project and at some point in the main

program calls the function representing the root behavior in the hierarchy. This begins automated execution and the translated framework will handle running the behaviors from this point onwards.

Since the behavior description in HLBL is independent of the platform, we were able to applied the framework of HLBL and B2C to various agent platforms, both robot and software agents. In particular, we have developed a behavior control for the AIBO [2] [3], and two software agents in simulated environment in a very short time. The framework also allowed us to quickly set up different behavior architectures for experimentation in the pursuit of the optimal one. This demonstrates the advantages of using the framework in agent programming.

2. Motivation and Background

While developing various agent and multi-agent systems, we often found ourselves reimplementing similar behavior synchronization and management code. When we wanted to experiment with different behavior architecture in order to find the optimal one for the current domain, we had to manipulate the native code of the implementation which was time consuming and error prone. In MONAD [1], one of our earlier works, we developed a scripting language that could represent hierarchical behavior architecture for multi-agent control. There was a run-time engine that would parse the script file to extract the behavior control routines. Using the scripting language had greatly reduced our time and effort in implementing a new behavior architecture as well as tweaking different parameters such as the dependency between behaviors, applicability conditions, or arbitration methods. We therefore decided to extend and enhance the scripting language in MONAD to a high level behavior-based language which has more expressive power, and develop a translator that can automatically generate the correspondent native code from the HLBL to facilitate the use of this language.

3. Behavior Hierarchy Representation and Execution

The behavior control routines for an agent will be implemented under the representation of a behavior hierarchy. Starting from the root of the hierarchy which serves as the entry point, the agent will execute the behavior and make the appropriate transitions based on the conditions of itself and the environment. The behavior hierarchy can be thought of as a finite state machine in which each behavior is equivalent to one state. Following is an example of a behavior hierarchy with 7 behaviors from B0 to B6. The labels SC, EC, and A denotes the set of starting, ending conditions and the action for each behavior respectively. C_i stands for the conditions for the agent to make the transition to behavior B_i. All those fields will be discussed further in the next paragraphs.

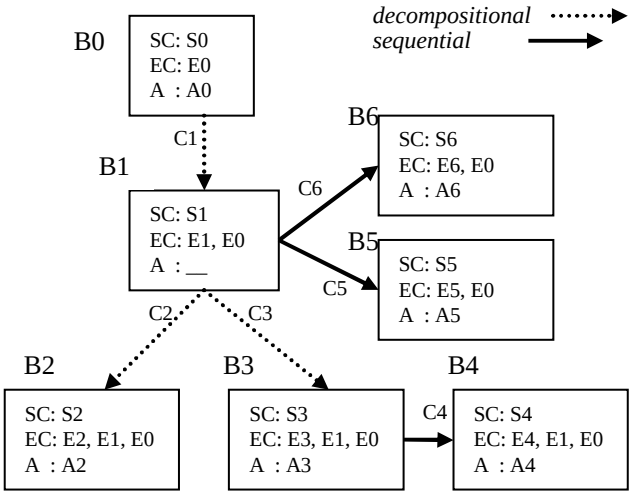

Figure 1. A behavior hierarchy.

Each behavior is associated with a set of starting conditions and a set of ending conditions. The starting conditions determine when a behavior is applicable and the ending conditions determine when the agent should stop executing the current behavior. Each behavior is associated with a goal that the agent tries to accomplish through the behavior. This goal is included in the set of ending conditions. As an example, the ending conditions of B2 are E2, E1, and E0. If any of those conditions evaluates to true, the agent will stop executing B2.

There are two types of transitions from one behavior: *sequential* transition to a following behavior and *decompositional* to a child one. The child-behaviors represent the alternative ways to accomplish the goal of the parent. The following behavior contains the next goal after the agent accomplishes the goal of the followed behavior, and it will share the same parent with the followed if there is any. In figure 1, B1 has two children B2 and B3, and since B4 follows B3, B1 is also the parent of B4.

When making a decompositional transition, the agent is considered as still trying to achieve the goal of the parent behavior. Thus, the set of ending conditions of a behavior contains as a subset the union of the ending conditions of its ancestors. In contrast, the agent has to accomplish the goal of the current behavior first before it can make a sequential transition. For an example, the ending conditions of B3 is E3, E1, and E0 since B0 and B1 are the ancestors of B3.

There can be an action for each behavior. The action can be any behavior but preferably an atomic execution behavior. The action will be carried out for every sense-

plan-act cycle that the agent stays in the current behavior until either the ending conditions meet or the agent can make a transition to the child-behavior. All the behaviors having no children must have an action. In figure 1, B1 does not have an action but B2 to B6 must have an action because they do not have any children.

Once the ending conditions of a behavior are met, the agent will stop executing the current behavior. If the goal of an ancestor of the current behavior has been accomplished or if the behavior does not have any following, the agent will return to the parent behavior. Otherwise it will make the transition to the following behavior.

A behavior can also have an initialization and a finalization function. The agent will call the initialization function when it first starts executing the behavior and the finalization function when it finishes the execution, returning to the parent behavior or making the transition to the next behavior.

While making a transition, either decompositional or sequential, if there is more than one option, the agent will have to make the choice between the available behaviors. Thus each behavior will be associated with one decision making mechanism called *resolution* for each transition it has to make. The designer can either define a resolution method using HLBL or use one such as random or alternating choices provided by B2C. Each resolution is essentially an ordered set of mappings from condition sets to choices. When the agent tries to make a decision, the first set of conditions satisfied will result in the associated choice being the result. For an example, from B1, the agent can make a sequential transition to either B5 or B6. Based on the conditions C5 and C6, the agent will decide which behavior it should make a transition to.

Below is the pseudocode for the agent to execute the behavior hierarchy:

```
1. B ← Root of the behavior hierarchy
2. Repeat until an ending condition of the root is true:
3.         Initialize (B)
4.         Repeat until an ending condition of B is true
5.                 Perform the action of B
6.                 If B has a child C available for transition,
7.                         B ← C
8.                         Go to 2
9.         Finalize (B)
10.        If B has a following F available for transition,
11.                B ← F
12.        Otherwise B ← parent of B
```

Figure 2. Pseudocode for the execution of the behavior hierarchy.

4. Example of Behavior Hierarchy Representation and Execution Model

In this section we will give an example of using the specified behavior hierarchy representation in implementing a program for the AIBO and how the program will be executed during run-time. This is part of the implementation we have done to evaluate the HLBL and the B2C system.

The AIBO is supposed to have the following behaviors:

- When it is placed on the ground, it will walk, trot, or run until it is lifted from the ground.
- If it is lifted straight up, it will wave its tail. If it is tilted to the left, it will turn on its middle left LED; if to the right, it will turn on its middle right LED.

Following is the correspondent behavior hierarchy:

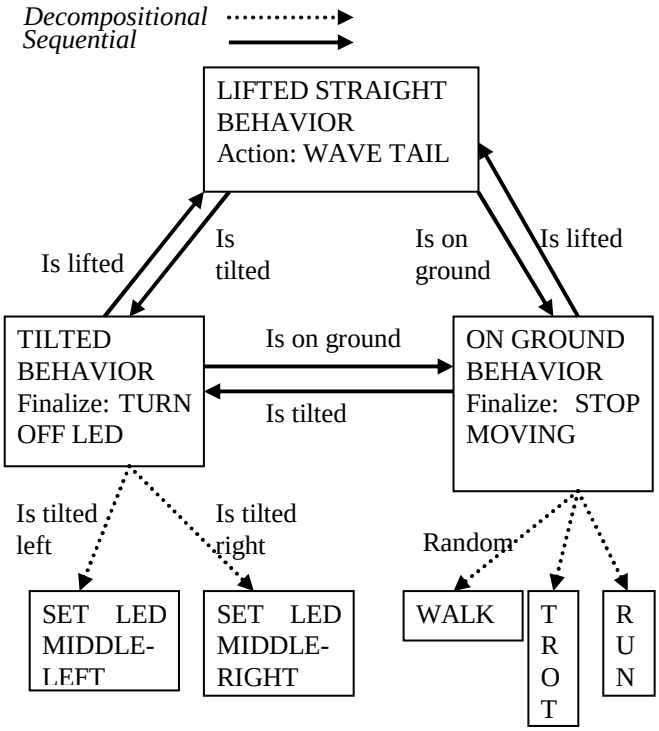

Figure 3. Behavior hierarchy for the AIBO.

The AIBO will start its execution from the "Lifted Straight Behavior". It will perform the action "Waving Tail" as long as the AIBO is in this behavior. There are two sequential transitions from this behavior: to "Tilted Behavior" when the AIBO is tilted, and to "On Ground Behavior" when the AIBO is actually on ground. Thus the set of ending conditions of this behavior is composed of the conditions "Is lifted" and "Is back on ground".

If the AIBO makes a transition to the "On Ground Behavior", we can notice that the behavior has three decompositional transitions beside the two sequential transitions. The two sequential transitions are: to "Tilted Behavior" when the AIBO is tilted, and to "Lifted Straight Behavior" when the AIBO is lifted up straight. The "On Ground Behavior" has three children: "Walk", "Trot", and "Run". When the AIBO starts executing "On Ground Behavior", since there is no action associated with this behavior, it will immediately make a transition to a random one among the behavior's children. Those children are atomic execution behaviors of the AIBO in the world. The "On Ground Behavior" also has a finalization function "Stop Moving." Thus it will call this function once the AIBO stops executing the behavior. The set of ending conditions is composed of the conditions "is lifted" and "is back on ground".

Similarly, the "Tilted Behavior" has two decompositional transitions: to "Set LED-Middle-Left" if the AIBO is tilted left and "Set LED-Middle-Right" if the AIBO is tilted right. Based on the condition the AIBO will make the appropriate transition. Since "Tilted Behavior" has a finalization function "Turn off LED", the AIBO will execute this function once it finishes executing this behavior. The two sequential transitions are: to "On Ground Behavior" when the AIBO is actually on the ground, and to "Lifted Straight Behavior" when the AIBO is lifted up straight.

5. Implementation of HLBL and Automated Code Generation

For an existing platform, there is a platform dependent library of condition checking functions and atomic execution behaviors. Given the API of the provided library, the designer will specify the agent's behavior architecture in a behavior description file using the HLBL.

The B2C system will then translate this file into correspondent C++ code. Once compiled, this code will yield an executable that can perform the specified behavior control routines.

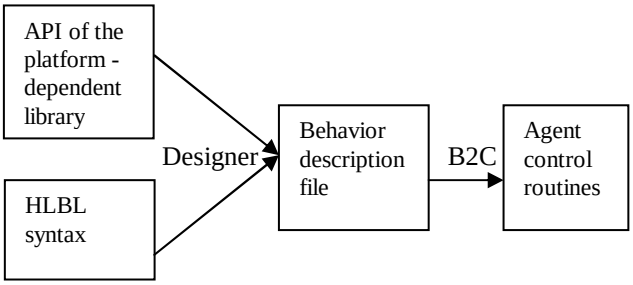

Figure 4. The process of generating agent control routines.

5.1 Platform-dependent library

The library is composed of a set of condition-checking functions and a set of behavior execution functions. The condition checking functions are boolean functions without arguments which compute some function of the current world state as perceived by the agent. As an example, the AIBO has sensors to detect if it is on the ground or not. Thus it has the boolean function "is_on_ground()" which will return true if the AIBO has its feet on the ground and false otherwise.

The set of behavior execution functions, which can be called at the end of each perception-thought-action cycle of the agent, support the actual execution of the agent in the world. These functions return nothing and also take in no arguments. For example, a behavior might have an execution function "Walk()" and thus as long as that behavior is active, the Walk() function will be called at every thinking cycle. The actual interior workings of the Walk() function are platform-dependent and as the agent merely calls the function Walk(), the actual execution of the function may involve sockets, threading operations, or any other code necessary to actuate the agent in the world.

5.2 Semantics of HLBL

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

behavior <behavior name>
The *behavior* keyword specifies that the following information from this point until an ***end*** keyword is used to describe a behavior

startswhen <condition>
The *startswhen* keyword is followed by a condition that uses a Boolean expression composed from the condition functions. This expression is called the applicability condition.

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

children <child 1> <child 2> … <child n>
The *children* keyword describes the list of child behaviors reachable from the current behavior. These child behaviors correspond to a decompositional transition from the current behavior. This keyword is optional and if omitted, there is only one way to accomplish the behavior

following<following1><following2>…<following>
The *following* keyword specifies the next same-level behavior that should be executed after the current

behavior. This keyword denotes sequential transition and it is optional.

child_resolution <resolution name>

The *child_resolution* keyword specifies the decision making method that should be run at the current behavior, when deciding which of its alternative child decompositions should be taken. This keyword can be present if and only if the behavior has some child.

following_resolution <resolution name>

The *follwing_resolution* keyword has similar meaning as the *child_resolution* keyword except that it is dedicated for sequential transitions only.

action <action name>

The *action* keyword is optional for behaviors with children but mandatory for the ones without. It specifies the atomic execution behavior that the agent needs to carry out within the current sense-plan-act cycle.

initialize <action name>

The *initialize* keyword is optional. It specifies the action that the agent needs to carry out once when it starts executing the behavior. This keyword is useful for resetting some internal state of the agent

finalize <action name>

The *finalize* keyword is similar with the *initialize* keyword. It is optional and can be used to specified the action that the agent needs to carry out once when it finishes executing the behavior.

resolution <resolution name>

The *resolution* keyword specifies that the following information until an ***end*** keyword is used to describe a method for the agent to make the choice among the behaviors that can be transitioned to.

cond <condition>

The *cond* keyword denotes the condition for one of the mappings in the behavior. It must be followed by a *choice* keyword

choice <behavior name>

The *choice* keyword follows a *cond* keyword to specify the behavior must be chosen if the followed conditions evaluated to true.

```
behavior Lifted_Straight
    endswhen is_tilted || is_on_ground
    following Tilted On_Ground
    following_resolution state_based
    action Wave_Tail
end

behavior On_Ground
    endswhen is_tilted || is_up_straight
    following Lifted_Straight Tilted
    following_resolution state_based
    children Walk Trot Run
    child_resolution RANDOM
    finalize Stop_Moving
end

resolution state_based
    cond is_tilted
    choice Tilted
    cond is_on_ground
    choice On_Ground
    cond is_up_straight
    choice Lift_straight
end
```

Figure 5. An excerpt from the behavior description file of the AIBO.

5.3 Auto-translation of the behavior description to code

B2C will take the behavior description file as input and generate a .cpp file with all the necessary links. Each behavior will be translated to a C++ function that has no argument and returns no value. A behavior makes transition to another one through the correspondent function call. The function begins with an if-statement that will terminate the function when the starting condition is not met. The body of the function is a while loop that will run until the ending condition is met. Inside the while loop, the action of the behavior will be called first and then the resolution function to get the name of the child for transition. The agent will call the function of the chosen child behavior if there is any.

After the while loop is the call to the finalization function. The agent will then call the resolution function for following functions. Upon receiving the result, the agent will call the function of the chosen following behavior if there is any; otherwise it will return to the parent.

As indicated in previous sections, the set of ending conditions of a behavior include the ending conditions of its ancestor. Therefore B2C will recursively propagate the ending conditions of one behavior to its children until no

new condition is added to a behavior. This guarantees each behavior will inherit all of its ancestor's ending conditions and if there is a loop of behaviors, the program will terminate once all the behaviors in the group have the same set of ending conditions.

The resolution method will be simply translated to a function that takes in no argument and returns the name of the chosen behavior. The function is composed of several if statement in the same order as the condition-choice pairs described by the designer. Each statement will return a string when its if-condition is met. At the end of the function, if no condition is evaluated to true, the function will return an empty string.

```
void call_function (string name) {
    if (name=="Lifted_Straight") {
        Lifted_Straight () ;
    };
    if (name=="On_Ground") {
        On_Ground () ;
    };
    if (name=="Tilted") {
        Tilted () ;
    };
}
string state_based () {
    if (is_tilted()) {
        return "Tilted";
    };
    if (is_on_ground()) {
        return "On_Ground";
    };
    if (is_up_straight()) {
        return "Lift_straight";
    };
}
void On_Ground () {
    do {
        if (is_tilted()||is_up_straight()) {
            break;
        };
        string child=  RAND_On_Ground_Res();
        call_function (child);
    } while (true);
     Stop_Moving();
    string next=  state_based();
    if (next!="") {
        call_function (next);
        return;
    };
}
```

Figure 6. An excerpt from the translated code.

To change the behavior hierarchy, the designer can simply edit the behavior description file. Once translated again, the new code will reflect the changes in the behaviors. Thus the designer can experiment different agent structures with little effort.

Previous version of the translated code:

```
void On_Ground () {
    do {
        if (is_tilted()||is_up_straight()) {
            break;
        };
        string child=  RAND_On_Ground_Res();
        call_function (child);
    } while (true);
    …
```

Updated version of the HLBL code:

```
behavior On_Ground
    endswhen is_up_straight
    following Lifted_Straight Tilted
    following_resolution state_based
    children Walk
    finalize Stop_Moving
end
```

Updated version of the translated code:

```
void On_Ground () {
    do {
        if (is_up_straight()) {
            break;
        };
        Walk();
    } while (true);
     Stop_Moving();
    string next=  state_based();
    if (next!="") {
        call_function (next);
        return;
    };
}
```

Figure 7. An example of how the translation reflects the changes of the behavior.

6. Applications of HLBL and B2C

We have applied HLBL and B2C in several scenarios. One of them is implementing behavior control routines for the AIBO. An example of the behaviors was shown in Figure 1. An excerpt of the correspondent behavior description file written in HLBL and the translated C++ code were shown in Figure 3 and 4, respectively.

Beside the program for AIBO, we have also developed two very different simulations utilizing HLBL and B2C.

The first simulation is a Maze Game in which the agent has to find the path from one given square in a maze to a target square. The game uses a very simple command line display. The platform dependent library is composed of functions for condition checking and atomic execution behavior such as checking if there is a wall on the left of the agent or moving the agent to the adjacent square on the right. There is one thread for the agent's behavior control routines and one thread for updating the world state, and the agent interacts with the world through appropriate function calls. The world state evolves in discrete time steps.

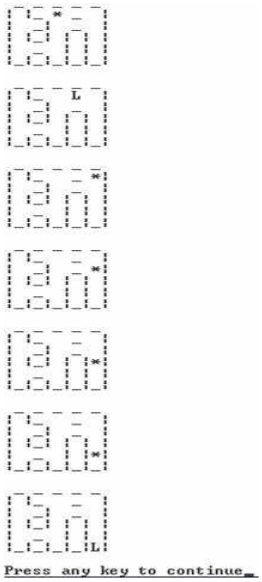

Figure 8. A screenshot of the Maze Game.

The second simulation is a Space Game in which the agent acts as a space ship trying to break down oncoming asteroids. In contrast to the other game, this game has a more complicated Direct3D display and runs as a continuous-time simulation. The agent uses sockets to interact with the world. The platform independent library includes functions and behaviors such as checking if there is an asteroid coming or moving one step to the left.

The agents in both games were running using compiled C++ code automatically translated from behavior description file in HLBL. HLBL significantly reduced the time require to implement the behavior control routines for these agents, and B2C allowed rapid integration of the routines into the underlying platform. Moreover, even though these two games are very different, the agents in these games still share a large part of their architectures, especially evident in the top-level behaviors. An example is given in figure 10. The agents in both games have very similar behavior "GoVertically". The only difference is their ending conditions. Thus, we were able to reuse several parts from the behavior architecture of the agent in the Maze Game when implementing the agent for the Space Game. This further facilitated the process of developing the agent.

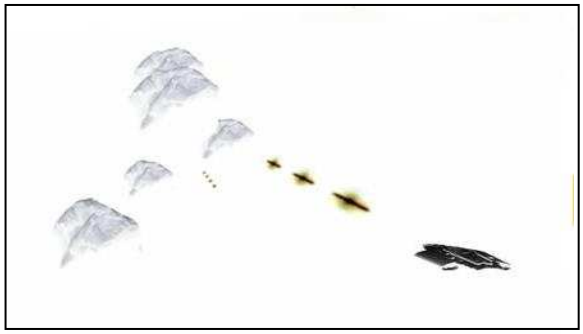

Figure 9. A screenshot of the Space Game.

With the behavior architecture implemented in HLBL, it is very easy to tweak different parameters such as the dependency between behaviors, the applicability and ending conditions, or the resolution methods. We could focus on programming the agents at the abstract behavior level without being burdened with the implementation details. Therefore we were able to experiment with many different behavior architectures in a short time in the pursuit of the optimal one. Our experience with using HLBL and B2C as described above demonstrates the advantages of using the framework to implement and experiment with agent behavior architectures.

```
behavior GoVertically
    startswhen  ShouldGoVertically
    endswhen CanWalkRight
    children MoveDown MoveUp
    child_resolution GoVerticallySplit
    following GoHorizontally
end
```

In the Maze Game

```
behavior GoVertically
    startswhen  ShouldGoVertically
    endswhen AsteroidOnRight
    children MoveDown MoveUp
    child_resolution GoVerticallySplit
    following GoHorizontally
end
```

In the Space Game

Figure 10. An example of reusing behaviors.

7. Summary

We have presented HLBL and B2C, the two key components of a new framework for agent programming. HLBL is platform-independent language with which one can design modular behavior architectures at the abstract behavior definition level. With this intuitive and flexible language, a designer can make rapid modifications to an agent's behavior architecture for experimentation and testing purposes. The automated code generation component, B2C, further facilitates the process of developing and experimenting with different behavior architectures by allowing designers to quickly integrate complex HLBL hierarchies directly into their agent's code. Future work will include expanding HLBL and B2C to allow an agent to execute more than one behavior in parallel and to support multi-agent systems.

8. Acknowledgments

We would like to thank Jared Go for his contribution to our early phase of this work.

9. References

[1] Vu, T.; Go, J.; Kaminka, G.; Veloso, M.; Browning, B. 2003. MONAD: A Flexible Architecture for Multi-Agent Control. In *Proceedings of the AAMAS'03,* p. 449 – 456, July 2003.

[2] Lenser, S.; Bruce, J.; Veloso, M. 2001. CMPack: A Complete Software System for Autonomous Legged Soccer Robots. In *Proceedings of the Fifth International Conference on Autonomous Agents,* May 2001.

[3] *CMRoboBits: Creating an Intelligent AIBO Robot* class http://www.andrew.cmu.edu/course/15-491/

[4] Kaminka, G.; Go, J.; Vu, T. 2002. Context-Dependent Joint-Decision Arbitration for Computer Games. In *Agent in Computer Games Workshop at ICCG'02,* July 2002.

[5] Brooks, A. 1986. A Robust Layered Control System for a Mobile Robot. In *IEEE Journal of Robotics and Automation* RA-2 (1) : 14-23

[6] Gat, E. 1991. ALFA: A Language for Programming Reactive Robotic Control Systems. In *IEEE Conference on Robotics and Automation,* 1991.

[7] Oza, N. A Survey of Robot Architectures.