Fusing Multiple Sensors Through Behaviors With the Distributed Architecture

Bradford A. Towle Jr. and Monica Nicolescu

Abstract— **This paper describes a new action selection mechanism that allows the robot controller to dynamically determine which goals (tasks) are most applicable in the current state of the environment. This allows the robotic controller to resolve conflicting goals that arise, allow multiple non-conflicting goals to run simultaneously, and dynamically switch to a new goal if the target for the goal has been found in the environment. This is accomplished by using a behavior based paradigm and allowing each behavior to calculate its own activation level (run level). The behaviors then use the activation level to compete for control in a new distributed control architecture to allow execution of multiple (and sometimes conflicting) goals on a robotic system.**

I. Introduction

The research in this paper will focus on the ability for a robot to execute multiple tasks, some with conflicting goals, in a rational fashion while still accomplishing all the goals in a timely manner. This is necessary for robots to handle multiple goals without constant human supervision. Imprecise actuators, limited sensor data, and a dynamic environment are problematic.

This paper presents a new architecture based upon the principles of behavior based system. This architecture incorporates a novel action selection mechanism that can accomplish multiple goal executions by allowing each behavior to calculate an activation level and compete for control of specific components on the robot. The activation level is continually updated and used by the robot to determine which goal is more efficient to execute at the time. This allows the robot to adjust to dynamic environments.

Three main contributions were derived through the research: Allowing for multiple goals, re-prioritizing goals based on outside stimuli and resolving conflicting goals through an auction based system. The architecture allows for multiple, non-conflicting tasks to be concurrently run on the robot by separating which actuators (if any) a behavior needs. The architecture then enforces an exclusive control of that actuator, thus allowing all behaviors which do not need that actuator to run.

The architecture can re-prioritize the goals when the sensor(s) detect a previously unknown target (goal state) nearby. This gives the robot the ability to dynamically replan. Lastly, the conflicting goals will be resolved by the architecture in a distributed fashion. The largest problem of allowing conflicting goals to compete occurs when the robot continually switches from one task to another (task oscillation) that achieves an undesired result. However, in the proposed distributed architecture, as one of the conflicting goals approaches completion, the activation level increases, thus resulting in a lower chance that another goal will have a higher activation level. Should the activation levels be exactly the same, which is statistically improbable, the system will choose the behavior which first requested control of an actuator. This same concept is used to indirectly arbitrate between user defined goals. Should a user specify two goals, the robot will estimate which goal is closer to being finished and attempt to complete it first.

A conceptual explanation is given on how the task selection mechanism works in section III. section III also details the new distributed architecture that allows multiple goals to function concurrently. Section IV shows the test results and give the proof of concepts for the three contributions of the paper: multiple goals, conflicting goals, and dynamic reconfiguration when a new goal is detected.

II. Related Work

There are several categories of action selection mechanisms. Two broad categories are arbitration and command fusion[10]. The first category, command fusion, combines multiple outputs of modules into a single output to send to the actuators. The second category, arbitration, chooses between different module outputs and selects one that is most appropriate to run. The arbitration mechanisms of interest is, priority-based, state-based, and winner-take-all.

Priority-based arbitration consists of modules (or behaviors) that are layered at different levels. Higher levels can inhibit or effect lower level modules because they have a higher priority. A good example of this would be subsumption [1]. This approach allows the designer to statically determine which goals are of higher priority; however, it cannot dynamically reconfigure the priority of a goal.

A second common arbitration scheme for task selection is State-Based arbitration. This method uses a finite state machine (FSM). The robot will be at a certain state and move to a new state depending on what outside stimuli occur[7]. This approach allows for more dynamic situations; however, the programmer has to specify all the state transitions so there are no conflicting goals nor can the robot run multiple goals at once. Another issue is all state transitions must be defined; therefore, this scheme limits the adaptability to dynamic environments.

The third approach to arbitration action selection mechanisms is winner-take-all, meaning the behaviors compete for exclusive control of the robot. Activation Networks [8] use a graph to contain goals which inject "activation energy" into the system. Each goal will determine how much "activation energy" based on sensors and states of the system. Each node

will then propagate this energy via three different types of links: successor, predecessor, and conflicter. The behavior with the highest activation energy wins control of the robot. The distributed architecture can handle conflicting goals as well as multiple goals where the activation network was designed to use the conflictor link to disable conflicting goals. This means the designer/programmer must know and be able to disable all behaviors that may conflict with the desired goal being programmed. This arduous task is not necessary with the distributed architecture because it uses a mutex schema to protect actuators.

This research also deals with multiple goals. Research has been done with cost benefit ratios[6] where the robot will dynamically re-plan when another high level goal wishes to run. This goal will then have total control unless an emergency arises or the environment changes. It is important to note that this method plans high level or abstracted task where the distributed architecture does this for each active behavior regardless of its abstraction level.

Work has also been done using semaphores (or mutexs) as a guard for system resources[11]. However, the distributed architecture also uses the semaphore/mutex idea as an auction house for behavioral competition. Where research up until now the semaphore/mutex idea simply is used to inhibit any behavior that would be conflicting with the current running goal.

The DAMN architecture[5] is very similar in design as the distributed architecture except it will fuse outputs from modules by allowing each behavior to vote for what it thinks the correct action for the robot is. The distributed architecture is winner-take-all, where the behavior with the highest activation level wins. This arbitration occurs every program cycle achieving a similar effect to the voting schema.

III. Approach

There are three main computational components of the distributed architecture which allow dynamic task selection. These components are goals, behaviors, and mutexes. These elements fit together to create a dynamic distributed system. Goals are objects which determine if a condition is met. If not, the goals are responsible for creating or linking to a behavior, functional component, which can complete the desired condition. Each cycle of the program consists of multiple execution steps. All goals specified by the user will determine whether or not they need to run. If they do, the behavior that is linked to the goal will compete for control of the mutex or allow any prerequisite goals to repeat the same procedure.

If the behavior can run, it will calculate how "far" the goal state is based on different parameters. For example, a behavior that uses a motor can use a Euclidean distance to goal, change in yaw and remaining time to calculate how "far" the desired goal state is from its current state. The behavior then requests control of a mutex. This mutex then scans all the state distances and determines if any particular distance is a maximum value for that distance type.

After all the behaviors and goals have been updated, the mutexes themselves share data to make sure they have up to date maximum state distances. Then, each mutex will request for each challenging behavior to calculate its activation level. The reason this must be done after all the behaviors have requested control is to ensure that the maximum values for each state distance are accurate. Each mutex will award the behavior, with the highest activation level control over the actuator. The final step activates each behavior and if the behavior has ownership, it can send commands to the actuator. It is important to note that a behavior could be designed to still provide some service without ownership of a mutex as long as it does not issue commands to an actuator or shared resource.

A. Goals

This project allowed a human to control the robot by specifying goals, not instruction sets. The goals are designed as a separate computational object from the behavior (or functional) object of the architecture. The goal represents a condition which needed to be met. While this condition was met, the goal would not attempt to use any behaviors. Since the goal is separate from the behavior, it can select which behavior is best suited to fulfill the requirement based on what actuators and sensors are available.

When a goal is created, it must either create a new behavior or link to an existing goal that can complete the desired task. It must also know when the task is complete and update itself to any changes that occur in the environment.

B. Behaviors

Before running the functional component of the behavior, each behavior updates the state distance and state variables it uses to determine the activation level.

The programmer can also use pre-requisites and co-requisite goals to create more advanced behaviors by telling the system what needs to be done before and while the behavior is executing. Abstract behaviors [9] can also be used to further the capability of the system. Abstract behaviors are behaviors that simply call other goals and do not actually command any actuators themselves. If the behavior has no unmet prerequisites, it will request control of any mutex it requires to run. Once the behavior has ownership of a mutex, it can send commands to the desired actuator. Also, the behavior does not actually have to use an actuator. It could simply read data from a sensor and return a value.

C. Mutex

Our proposed approach for task selections was to use the concept of a mutex to guard resources on the robot. In operating systems and parallel computation theory, a mutex is used to enforce mutual exclusion of code for certain critical areas. For example, if two threads use a shared memory space, they usually cannot access it at the same time. The mutex in this distributed architecture enforces that only one behavior at a time can control the resource it is guarding. For the research purposes, a camera, gripper, and motor mutex

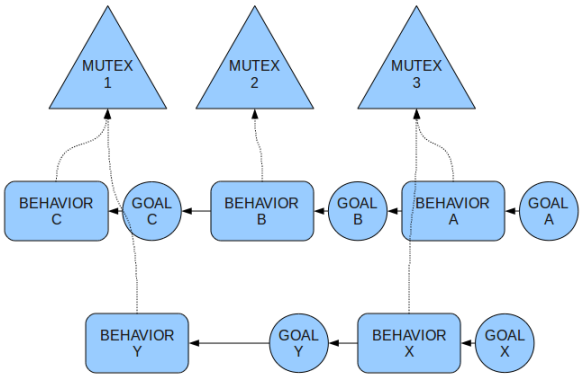

Fig. 1: Goal A and X are both user specified goals. The remaining goals are prerequisites. Behaviors X and A will compete for Mutex 3. Behaviors C and Y will compete for Mutex 1

were used. These entities would record all the behaviors which requested control and then award ownership of the resource to the behavior with the highest activation level. In other words, the mutex acts as an auction house.

An important attribute of most robotic architecture is the capability of being distributed across multiple processors. When addressing the mutex concept, it appears that the rule of distribution is broken. However, consider any parallel system which requires a hardware resource. The mutex which guards that resource has to be reachable by any process that requires it. Also, it makes sense to place the mutex on the processor in which the piece of hardware is attached. The architecture itself is distributed. A programmer could divide separate behavior chains or even individual goal/behavior pairs amongst any number of processors. However, the processor where the actuator was controlled would still need a mutex to protect it. Thus, using the mutex as an auction house does not break the distributive nature of the architecture.

D. Calculating Activation Levels

The action selection mechanism of the distributed architecture contains two possible metrics which can be combined to compute an activation level of a behavior. The first metric is a state distance to a desired goal state. This state distance would be compared with the maximum distance for that particular type, and the smallest ratio would be given the highest priority. Thus, increasing the activation level of that particular behavior the most. As mentioned before, this state distance is not necessarily Euclidean. There are several variables such as time which are non-spatial.

The second metric used is a state counter. When trying to calculate an activation level, it became apparent that there could be other possible variables that could contribute to the importance of the behavior. Mathematically, the only other variable type that was feasible to incorporate and still allow a versatile degree of programming was a counter. By adding a state counter, the behaviors were now able to track how many times they occured over time. The main difference is the state counter would increase the activation level based on how high (or low) it was versus the state distance which would increase the activation level based on how much smaller it was from the maximum value of that type (explained in the following two sections). The state counter is divided by a scaling factor that the system changes (see section III-D.2) and added to the activation level.

1) State Distance Integration: Each running behavior reads the sensor data it requires. (For example: laser range finder, odometry, camera blob-tracker, etc.). From these data, the behavior tries to determine the state distance to the desired goal state for each state variable. If a state distance cannot be determined or estimated, the behavior sets the value to -1 specifying that it cannot use that particular state variable for the activation level calculations. Again the behaviors only consider the state variables in terms of what they control. For example, a camera controller would utilize camera pan and camera tilt while not considering the location of the robot or any readings from the gripper.

There were three mutexes developed for each actuator in order to prove this concept: gripper, motor, and camera. First, each behavior is aware of the state variables it uses. An active behavior should be able to generate a state distance to the goal state for each state variable it uses (it can be a spatial distance or state distance). When the behavior requests control, the mutex looks through the behavior distances and determines if any are the maximum for that type. Each behavior needs to request the maximum values from the mutex it is competing for. Then, each distance is divided by the maximum value of that variable type. The result is a normalized ratio (between 0 and 1). Once these values are found for each state distance in the behavior, they are all averaged together and the result is a normalized activation level (2).

$$F(x) = \sum_{i=0}^{Np} \frac{max_K(V_{k,i})) - V_{b,i}}{max_K(V_{k,i})} \tag{1}$$

Which simplifies to:

$$F(x) = Np - \sum_{i=0}^{Np} \frac{V_{b,i}}{max_K(V_{k,i})} \tag{2}$$

$$NormalizedF(x) = \frac{F(x)}{V} \tag{3}$$

$$\tag{4}$$

$F(x)$ =State Distance Component of the Activation level.
$V_{K,i}$ is the maximum value for the i^{th} variable which is stored in the mutex.
V is total number of active behaviors.
b is the behavior that is requesting control.
Np is the total number of different parameter types.
K is a set of all challenging behavior's parameter of type i.

With this technique, the sensor data are transformed into state distances, which are then translated into comparable ratios with any other behavior. This technique only works

with state distances, where a max value of the same type of distance can be found.

2) State Counter Integration: The second metric of the action selection mechanism is a general state counter. For example, a counter was used to track the number of consecutive failed attempts at gaining access to the mutex.

The state counter cannot be calculated the same way as the state distance. If all state counters values were close in proximity, they would exhibit little influence on the activation level, especially if they all contained high values. This is because the state distance algorithm calculates a ratio and measures the difference between the state distance and the maximum distance for that state distance type. Likewise, in situations where the counters are relatively low, a single increment could influence the activation level substantially even though it has only incremented once.

In order to incorporate counters into the run level, the system must know what to scale the counter by. In other words, divide the number of the state counter by the scalar value to get the ammount you must add to the activation level. This scalar value can be arbitrarily assigned or the value can be estimated by the different outputs from equation 4. For example, to make the activation level summation count up .01 every time the counter increments the resolution would be 100. Using the following piece-wise equation, a counter can be incorporated into the distance ratio (see equation 5).

$$Y(x) = \begin{cases} count/scalar & \text{if} count < res, \\ 1.0 & \text{if} count \geq res. \end{cases} \quad (5)$$

$count$ is the counter that is being incorporated.
$scalar$ is the scalar value (normally a power of ten) in which to divide by.
For the purpose of this project the scalar value was set to 100. The result of equation 5 would be added to the sum of the ratios from equation 4. Also, the number of active variables would need to be increased in order to keep the normalized ratio (less than or equal to 1). After finding the normalized value for the counter, it must be incorporated with the measurement value to calculate the activation level. The final equation for a behavior's activation level then looks like equation 6.

$$Activation\_Level = \frac{F(x) + Y(x)}{Np + Nc} \quad (6)$$

Nc is the number of counters that are being incorporated into the run level ratio.
Np is the number of active measurements being incorporated into the run level ratio.
$Y(x)$ is the result from equation 5.
$F(x)$ is the result from equation 2.

IV. Results

In order to test this architecture, the Player/Stage robotic simulator was used to test the task selection mechanism. Three scenarios were conceived in order to prove that the robot could handle multiple conflicting goals, incorporate time into the selection mechanism, and finally re-prioritize a goal should the desired target of that goal be detected in the environment.

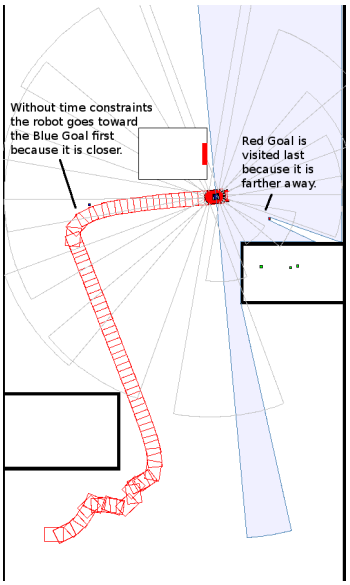

Fig. 2: The robot's path as it attempts visit two goals.

Competing Behaviors	Red Target Blue Target
Mutex which is being competed for	Motor
State variables used to find activation levels:	Euclidean distance robotic yaw Time left until start Denial counter

TABLE I: Competing Behaviors and the parameters used.

A. Multiple Conflicting Goals

The red and blue targets shown in Figure 2 need to be visited by the robot in order to complete the test. For a breakdown of which behaviors were tested please refer to Table I. From the robot's starting location the blue goal is closer therefore this goal's behavior has a higher activation level. This causes the robot to move to the blue goal first then moves to the red. From Figure 3, oscillation can be observed when the test first begins, however, the architecture quickly converges to visiting the blue target first.

The smart wander behavior was used by the "go to" behavior as a prerequisite goal. If the robot came within a certain distance of a wall the "go to" behavior would activate a smart wander goal. This smart wander was only activated if the go to behavior had gained control of the mutex. Therefore it did not compete with the parent behaviors (Go to red target and go to blue target)

B. Multiple Conflicting Goals With Time Constraint

This scenario is set up like the first scenario with a blue and red target. However, this time, the red goal uses a much smaller "time to start" value to demonstrate the ability for this system to prioritize goals on other basis besides

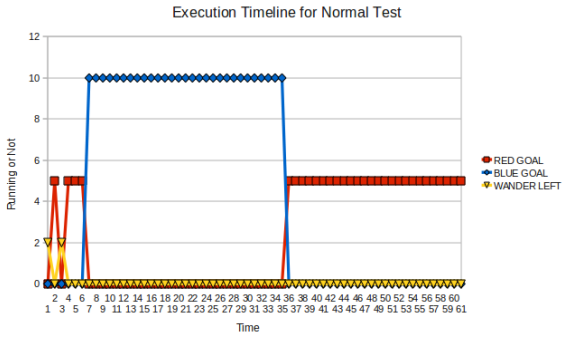

Fig. 3: The Time Line of execution for the "visit two goals" test.

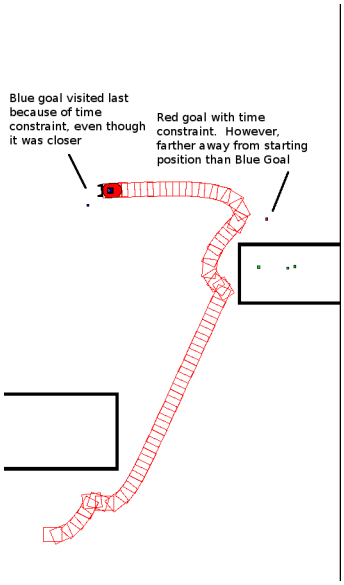

Fig. 4: The robot's path to both a time critical goal and a non-time critical goal

Euclidean distances. See Table I for a list of the behaviors and parameters (They are the same as the first test). Figure 4 details the path taken by the robot going to the red goal first due to its more critical time limit then to the blue goal. As seen in Figure 5, oscillation occurred between the red and blue goal around the middle of the test. This was caused by the robot making a small detour around a wall and coming very close to equal distances between the two targets. However, with a significant difference in the time constraint, the robot returned to trying to finish the red goal. If the robot had moved into close proximity of the blue goal, it may have chosen to finish the blue goal before finishing the red goal purely for convenience sake.

C. Dynamic Re-planning When A New Goal Becomes Activated

The last scenario duplicates the original test but also requires the robot to turn and face a green target with blob tracker should it encounter a green object. Figure 6 gives the path the run and II details the behaviors and goals used. One main difference between this test and the orginal test is

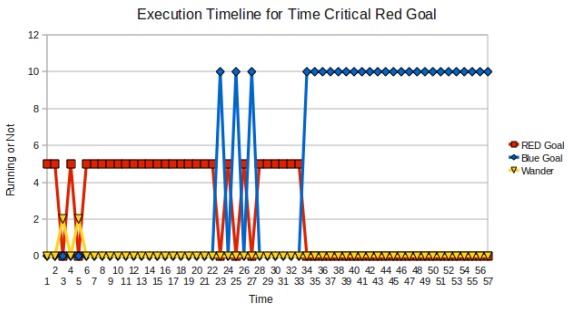

Fig. 5: The time line of execution for the "time critical goal"

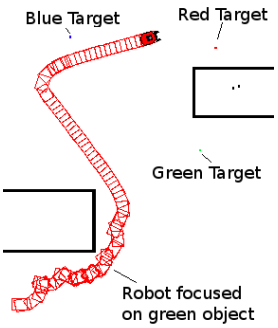

Fig. 6: The robot's path while discovering a new goal

the "Center on Color" object was used. Once the robot is in range of the green object it centers itself on the target before moving on to the next goal. After the green goal has been completed, the blue goal is closer to the robot. The red goal does not have a time constraint; therefore, the blue goal is achieved first. Figure 7 displays the timeline of execution for this task. This system used two other behaviors for swiveling the camera and centering the camera on the target. These two behaviors do not compete for the motor mutex therefore they are not included in the table and graph. This proves that the distributed system allows multiple non-conflicting goals to be activated at once.

V. Future Work

Abstraction levels are used in robotics to solve complicated tasks with smaller, easier components. However, it is difficult to define what the levels of abstraction are. Even a well formed criteria for abstraction can have ambiguity [12][4]. The distributed architecture provides three levels of abstraction without any ambiguity. The lowest level consists

Competing Behaviors	Go to Red Target Go to Blue Target Center Robot on Green Object
Mutex which is being competed for	Motor
State variables used:	Euclidean distance Change in robotic yaw Time left until must start Denial counter

TABLE II: Competing Behaviors and the parameters used.

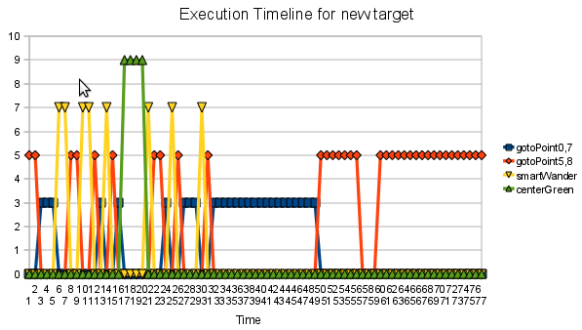

Fig. 7: The time line of execution for the "goal discover test"

of any goal that was not explicitly specified by the user. The second layer would consist of any user specified goal. The highest layer would be a collection of all the user specified goals. These three definitive levels of abstraction could provide standardization allowing different development teams the ability to create compatible software.

Another area of research is human and robot teams trying to efficiently allocate tasks with each other for the purpose of accomplishing a goal more efficiently [2][3]. This is generally used to find the most efficient work plan and take full advantage of limited field time (i.e. astronauts on Mars). Finally, any system that can dynamically change its goals is not necessarily "user friendly". Thus, considerations must be made for a human interacting with any robot using this paradigm. The problem becomes more difficult if you consider the multi-agent domain where the human must now control multiple robots at the same time.

VI. Conclusion

This architecture provided three main points: The robot has the capability of achieving multiple non-conflicting goals. The architecture allows for goals to be dynamically detected from the enviroment. The architecture can resolve conflicting goals by determining which has a higher priority. This architecture provides the robot with the capacity to run multiple behaviors which do not require the same actuator. This is done by using a mutex to protect robotic actuators. This allows the robot to use more of its non-conflicting capabilities at once and provides more efficient task allocation. Secondly, the distributed architecture allows for discovering new goals and making logical decision as to whether or not it needs to be run. This allows the distributed architecture to adapt to a dynamic enviroment by re-prioritizing the active goals. Finally, the distributed architecture will use an activation level to prioritize the goals. This activation level will then be used to compete for control of an actuator against other behaviors. As the robot approaches the goal state, the activation level of the behavior of the goal will increase, thus reducing the chance of goal oscillation. With these three contributions, the robot can re-prioritize tasks which may become activated at any time in a dynamic environment.

References

[1] R.A. Brooks. A robust layered control system for a mobile robot, 1985.

[2] G. Dorais, R.P. Bonasso, D. Kortenkamp, B. Pell, and D. Schreckenghost. Adjustable autonomy for human-centered autonomous systems. In *Working notes of the Sixteenth International Joint Conference on Artificial Intelligence Workshop on Adjustable Autonomy Systems*. Citeseer, 1999.

[3] Y. Endo, D.C. MacKenzie, and R.C. Arkin. Usability evaluation of high-level user assistance for robot mission specification, 2002.

[4] A. Fereidunian, M. Lehtonen, H. Lesani, C. Lucas, and M. Nordman. Adaptive autonomy: smart cooperative cybernetic systems for more humane automation solutions. In *IEEE International Conference on Systems, Man and Cybernetics, 2007. ISIC*, pages 202–207, 2007.

[5] E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *Proceedings of the National Conference on Artificial Intelligence*, pages 809–809. Citeseer, 1992.

[6] R. Goodwin and R. Simmons. Rational Handling of Multiple Goals for Mobile Robots. 2004.

[7] J. Kosecka and R. Bajcsy. Discrete event systems for autonomous mobile agents. *Robotics and Autonomous Systems*, 12(3):187–198, 1993.

[8] P. Maes. How to do the right thing. *Connection Science*, 1(3):291–323, 1989.

[9] M.N. Nicolescu and M.J. Matarić. A hierarchical architecture for behavior-based robots. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 1*, pages 227–233. ACM New York, NY, USA, 2002.

[10] P. Pirjanian. Behavior coordination mechanisms-state-of-the-art. *Institute for Robotics and Intelligent Systems, School of Engineering, University of Southern California, Tech. Rep. IRIS-99-375*, 1999.

[11] J. Rosenblatt and C. Thorpe. Combining multiple goals in a behavior-based architecture. In *Proc. IEEE Conference on Intelligent Robots and Systems*. Citeseer, 1995.

[12] TB Sheridan and C. MIT. Human centered automation: oxymoron or common sense? In *IEEE International Conference on Systems, Man and Cybernetics, 1995. Intelligent Systems for the 21st Century.*, volume 1, 1995.