In *Working Notes of the AAAI Fall Symposium on Parallel Cognition*
pages 27-34, North Falmouth, MA, Nov 3-5, 2000.

Extending Behavior-Based Systems Capabilities Using An Abstract Behavior Representation

Monica N. Nicolescu and Maja J. Matarić
Computer Science Department
University of Southern California
941 West 37th Place, Mailcode 0781
Los Angeles, CA 90089-0781
monica|mataric@cs.usc.edu

Abstract

Behavior-based systems BBS have been effective in a variety of applications, but due to their limited use of representation (sentence or logic-like structures) they have not been used much for more complex problems involving sequences of behaviors and they have been typically constructed by hand for each task. In this paper, we present an abstract behavior representation that allows for automatically specifying behavior networks that encode complex behavioral sequences, based on a given set of underlying behaviors, and avoids customized behavior redesign while accommodating the specifics of a new task. The representation, obtained by separating behaviors into two classes, abstract and primitive, allows BBS to generate and maintain complex plan-like strategies as well as switch them at run-time, without any need for behavior redesign and/or recompilation. To validate the described representation we have performed two object delivery tasks, involving behavior conflicts and various initial conditions, using a Pioneer2 DX mobile robot.

Introduction

Behavior-based control (Arkin 1998, Matarić 1997) has become one of the most popular approaches to embedded system control both in research and in practical applications. Behavior-based systems (BBS) employ a collection of concurrently executing behaviors, processes connecting sensors, effectors, and each other. An important property of BBS is their ability to contain state, and thus also construct and use distributed representations. This ability has been underused, so BBS are yet to be explored and extended to their full potential. In this paper we address two current limitations of such systems, both having to do with the use of representation, which motivate our work.

The first motivation is the fact that behaviors lack the abstract (symbolic, logic-like) representation that would allow them to be employed at a high level, like operators in a plan. Behaviors are typically invoked by buit-in reactive conditions, and as a consequence, BBS are typically unnatural for, and thus rarely applied to complex problems that contain temporal sequences. Since there is no intrinsic limitation within BBS expresivity that prevents this capability, in this paper we present a method for taking advantage of it.

The second, and related, motivation for this work is that the vast majority of behavior-based systems are still designed by hand for a single task: the lack of abstract representation prevents automatic generation of BBS. Also, behaviors themselves, once refined, are usually reused by designers, enabling the gradual accumulation of *behavior libraries*. Unfortunately, the remainder of the system that utilizes such libraries is usually constructed by hand and involves customized behavior redesign in accordance with the specifics of any new task. Our aim is to conserve the robustness and real-time properties of behaviors and to develop a behavior representation that would support automatic generation of BBS and behavior reuse for multiple tasks (at least within a class of related tasks) while avoiding behavior redesign and even recompilation when switching to a different task.

Attempts to solve these issues have resulted either in hybrid architectures, or in behavior-based architectures that only partly address the above problems. We propose a representation that does not alter the nature of behavior-based systems and neither changes the representation nor the time-scale. We present a detailed discussion of the differences between the existing architectures and ours in the Related Work section.

The abstract behavior representation that we introduce is based on behaviors developed for any one or more specific tasks. It is critical that the practical, robust behaviors come first, and the representation is derived from them. This stands in sharp contrast to approaches that employed high-level sensors and operators assuming that the low-level controller will provide whatever information and action was needed by a high level planner (see the Related Work section).

The abstract behaviors are used to specify one or more tasks, in the form of behavior networks, which can be generated not only by hand but also automatically, depending on task complexity. Any single network represents a task-specific BBS, much like standard BBS. However, the components of the networks are general, allowing for behavior reuse both off-line (for system specification) and on-line (for system adaptation to a new task or directive).

In the remainder of the paper we first describe the notion of abstract behaviors, then introduce the behavior network construct that uses them to represent general strategies and plans. We describe how these constructs can be defined, and finally, we validate them in real robot experiments. We demonstrate two different versions of a mobile robot object delivery task, and provide experimental results using a Pioneer robot. We end the paper with a review of related work, our continuing work on this topic, and conclusions.

Adapting Behaviors for Representation

BBS behaviors typically consist of a collection of rules, taking inputs from sensors or other behaviors in the system, and sending outputs to the effectors, or other behaviors. The inputs determine the activation level of a behavior: whether it is on or not, and in some systems by how much. These are the activation *conditions* for behavior execution. For the purposes of the representation, we distinguish the following two types of activation conditions (behavior preconditions):

- *world preconditions* - conditions that activate the behaviors based on a particular state of the environment.
- *sequential preconditions* - task-dependent conditions that must be met before activating the behavior. These are often postconditions of other existing behaviors, which allow for the description of complex temporal sequences.

In standard BBS behaviors, both types of preconditions are tested together, and without discrimination, thus hard-coding a particular solution. To change tasks and goals, one often makes the most changes to these preconditions, while much of the rest of behaviors remains unchanged. We achieve the ability to manipulate and change these conditions at an abstract representation level, separate for the behavior repertoire/library, by introducing abstract behaviors.

With those, behaviors are treated as high-level operators, and without loss of robustness can be employed to generate various strategies or plans for specific tasks. While classical planning requires a specific initial state, BBS provide general controllers that can handle a variety of initial conditions. With the use of abstract behaviors, we generate networks that are BBS, being triggered by whatever condition the environment presents.

In their operation, behaviors individually or as a group achieve and/or maintain the goals of the system, thus achieving the task. This methodology lends itself to the construction of highly effective special-purpose systems. This is thus both a strength and a weakness of the approach. In order to lend generality to a given system, we first looked for a way to make the behaviors themselves more general, while still assuring that they would achieve and/or maintain the goals for which they are designed.

The key step in adapting specialized behaviors to more general use is in the separation of the activation conditions from the outputs or actions. By separating those conditions from the actions, we allow for a more general set of activation conditions for the behavior's actions (Figure 1). While this is not necessary for any single task, it is what provides generality to the system for multiple tasks. The pairing of a behavior's activation conditions and its effects, without the specification of its inner workings, constitute an *abstract behavior*. Intuitively, this is simply an explicit specification of the behavior's execution conditions (i.e., preconditions), and its effects (i.e., postconditions). The result is a an abstract and general operator much like those used in classical deliberative systems (Fikes & Nilsson 1971). The behaviors that do the work that achieves the specified effects under the given conditions are called *primitive behaviors*, and may involve one or an entire collection of sequential or concurrently executing behaviors, again as is typical for BBS.

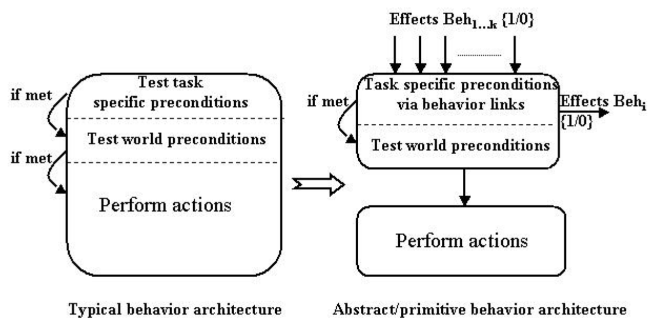

Figure 1: Adaptation of typical behaviors for abstract representations

Abstract and primitive behaviors can both be quite complex, just as they are within any system embedded in an environment. The abstract behavior conditions, as in any BBS, are typically far from low-granularity states, but are instead abstracted, either by hand or through a generalization process. If they were not, the benefits of using behaviors as a high-level representation would be lost. Similarly, the primitive behaviors are no lower level than standard BBS behaviors, meaning they are typically time-extended sequences of actions (e.g., go-home), not low-granularity single actions (e.g., turn-left-by-10-degrees).

Behavior networks then are a means of specifying strategies or general "plans" in a way that merges the advantages of both abstract representations and behavior-based systems. The nodes in the networks are abstract behaviors, and the links between them represent precondition and postcondition dependencies. The task plan or strategy is represented as a network of such behaviors.

As in any BBS, when the conditions of a behavior are met, the behavior is activated. Similarly here, when the conditions of an abstract behavior are met, the behavior activates one or more primitive behaviors which achieve the effects specified in its postconditions. The network topology at the abstract behavior level encodes any task-specific behavior sequences, freeing up the primitive behaviors to be reused for a variety of tasks. Thus, since abstract behavior networks are computationally light-weight, solutions for multiple tasks can be encoded within a single system, and dynamically switched, as we demonstrate in our implementation.

In the next sections we present the structure and functionality of abstract and primitive behaviors, then the construction of networks and their use.

Abstract Behaviors

As mentioned above, adapting specialized behaviors to general use requires a separation between the execution conditions and actions. We group these execution conditions and the behavior effects into abstract behaviors which have the role of activating the primitive behavior(s) that achieve the specified effects. In order to include behavior effects into the abstract representation we provide abstract behaviors information about the behavior's goals and a means of signaling their achievement to other behaviors that may utilize (and in fact rely on) these effects.

An important characteristic of our behaviors that makes our architecture even better suited for high-level, complex tasks, is that they are parameterizable. The behavior goals are represented as "predicate-like" structures in terms of the behavior parameters. The quotes above are used to stress that the effects are abstracted environmental states (continuously computed from the sensors) and not high-level symbols that are not grounded in perceptions. Thus, our behaviors become even closer, in terms of functionality, to the abstract operators used in symbolic architectures, allowing for multiple parameter bindings and therefore multiple and different goals for only one behavior, while still maintaining the real-time properties of behaviors.

The state of a behavior's goals (achieved or not) is fed into a behavior output connected to all the behaviors that require that condition to be true before they can become active. In this way, the information about the task-specific preconditions can be automatically obtained from the behavior network precondition-postcondition dependencies and dynamically changed (by simply rearranging the links) if networks need to be switched at run-time. This allows for obtaining multiple solutions while using the same behaviors and maintaining the goals for which they have been designed.

As with operators in a plan, behaviors can undo each other's actions while trying to achieve their own goals (Chapman 1987). In BBS, such undesirable competition is typically handled either by mutually-exclusive behavior execution conditions, or by the behavior coordination mechanism (Pirjanian 1999). In this work, we take the former approach, and use inhibition between behaviors, a common BBS tool, to prevent destructive competition. This methodology directly fits into the behavior network representations: the network topology also includes inhibitory links between competitive behaviors.

In our implementation behaviors run at a predefined rate at which they continuously check or send their inputs and outputs. In a discrete implementation, single activation and deactivation messages could be used per behavior, but this would not be as robust. Our system, as most BBS, uses continual messaging, in order to remain reactive to any changes that may occur (in the environment, the preconditions, etc.)

Informally, an abstract behavior sends an activation signal to its associated primitive behavior(s) if it is being used in the current controller, it is not inhibited by another competitive behavior, if its own postconditions are not yet met, and if the preconditions of its "redecessor behaviors" are met. More formally, the computation/processing performed by an abstract behavior (see Figure 2) is as follows:

1. If the *UseBehavior* input signals that the behavior is used in the current network controller (i.e., if this behavior is a part of the currently executing task), it continues with the next step. Otherwise, it returns.
2. Using the information from the sensors it computes the values of the predicates representing the behavior's goal, derives from them the state of achievement/unachievement of that goal and writes this value to the $Effects$ output, in order for it to be accessible for any behaviors that may rely on those particular conditions. If the goal is achieved the behavior returns. Otherwise, it continues with the next step.
3. Check the *Inhibit* input to see whether the behavior is inhibited. If false, go to the next step, otherwise return.
4. Check the *Continue* input. If true, set the *Active* output to 1 and return. The *Active* output has the role of activating/deactivating one or more primitive behaviors to which it is connected.
5. Check the effects of all the behaviors on which the current behaviors depends (the $Effects_{1...k}$). If all those behaviors have their goals met, set the *Active* output to 1, activating the corresponding primitive behavior(s) , otherwise set it to 0. Our architecture allows for specifying activation conditions that are disjuncts of several preconditions, which in turn are conjuncts of other conditions. Thus:

$Active = Precondition_1$ OR $Precondition_2$ OR ... OR $Precondition_n$, where

$Precondition_i = Effects_1$ AND $Effects_2$ AND ... AND $Effects_m$

Primitive behaviors

Primitive behaviors (see Figure 2) are activated by abstract behaviors via the *Active* input; they are the behaviors that actually achieve the goals represented by the abstract behaviors.

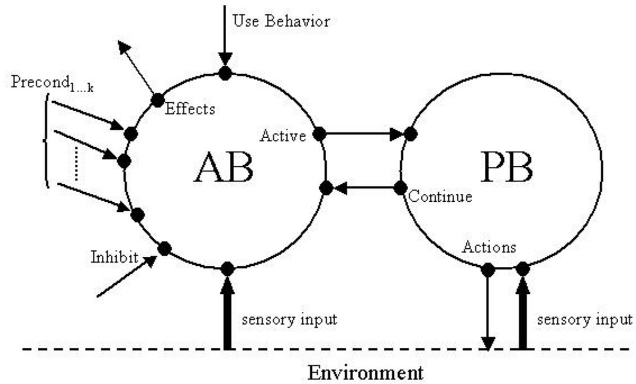

Figure 2: Structure of the inputs/outputs of an abstract and primitive behavior.

Primitive behaviors use sensory information in order to compute the actions sent to the system's effectors via the *Actions* output. The *Continue* output is used to notify the corresponding abstract behavior that the execution of the behavior is not finished yet so that the abstract behavior continues to send activation. This output is used only in situations in which it is important that the execution of the primitive behavior not be interrupted, such as those caused by transience of sensory data. In these cases, it is necessary to extend the execution of the behavior until its completition. In all other situations, the abstract behavior can stop sending its activation at any time, according to its current conditions.

Behavior Network Construction and Execution

The purpose of our abstract representation is to allow behavior-based systems to benefit from two important characteristics of symbolic systems.

First, in order to allow BBS to perform complex temporal sequences, we have embedded in the abstract behaviors the representation of the behavior's goals and the ability to signal their achievement through output links to the behaviors that are waiting for the completition of those goals. The connection of an $Effects$ output to the precondition inputs of other abstract behaviors thus enforces the order of behavior execution. The advantage of using real behaviors can be seen again when the environment state changes either favorably (achieving the goals of some of the behaviors, without them being actually executed) or unfavorably (undoing some of the already achieved goals): since the conditions are continuously monitored, the system continues with execution of the behavior that should be active according to the environmental state (either jumps forward or goes back to a behavior that should be reexecuted).

Second, we intend to use the above behavior architecture to automatically generate behavior networks for any given task. As we described above, the abstract behaviors specify the goals in a "predicate-like" form on the behavior's parameters, which would make them suitable for use with a general purpose planner in order to obtain a solution for a given task. However, in order to provide a solution, a planner would require the complete specification of the initial state of the system. Since our networks rely on real behaviors which can handle a variety of initial conditions, we want to construct a behavior network generator that would take advantage of this characteristic and would build a single BBS controller, the same as a human designer would do.

The process of behavior network generation that we describe bellow is work under development, and we present it here only at a high level.

As is known, behavior goals can be of *achievement* or *maintenance* and for the purpose of our algorithm we will also differentiate between these. The process of generating a behavior network for a given goal begins by backtracking, from that goal and adding to the behavior network all behaviors that can achieve all or some of the conditions of the goal (we mark those behaviors by setting their $UseBehavior$ input to 1). At this same step, behavior parameters are bound to values that would allow them to achieve the goal. For the added behaviors that achieve maintenance goals links are added from their $Effects$ output to the precondition inputs of all the other behaviors in the library. Of course, in the end only the behaviors that have been added to the network by the described process will be used. This is necessary in order to enforce that maintenance goals, if undone, would immediately activate the execution of the behaviors that achieve them. From this point, the algorithm continues with the same mechanism, considering adding to the network the behaviors that achieve the preconditions of the previously added behaviors. The process continues until each behavior in the network has one or more behaviors that can achieve its preconditions.

Our algorithm also must provide the resulting behavior network topology with the set of necessary inhibition links between conflicting behaviors. We have considered the follwoing types of behavior conflicts so far:

1. Behaviors that undo each other's actions and, as a result, can become active at the same time;
2. Behaviors that can be active at the same time and also may undo each other's actions;
3. Behaviors with complementary goals-preconditions, that can generate loops within the controller.

The first type of conflict can be easily solved by implicit inhibition links between the involved behaviors. These conflicts can be detected in the network construction process described above and simply solved by connecting the *Active* output of each of the behaviors involved to the *Inhibit* input of the others. Thus, when one of the behaviors is active, the other one will be inhibited.

However, for the second and third types of conflicts we have to use explicit inhibitions/de-inhibitions and the detection of these conflicts can only be done at run-time, since it depends on the current state of the environment.

The process described above provides a more general solution than a planner would do, due to the fact that it relies on real behaviors. Figure 3 shows the behavior networks created by hand for the delivery task that we are further demonstrating.

At initialization, according to a network specification (see Figure 3), each behavior receives a list of its precursor behaviors used to specify the network topology (i.e. the Effects-Precondition links). Once the network is constructed, whichever abstract behavior has all of its preconditions met activates its corresponding primitive behavior(s), thus initiating the system. The system will proceed by having both sets of behaviors in concurrent operation. Abstract behaviors activate (and sequence) primitive ones, and in turn, primitive behaviors can notify abstract ones about their execution. Note that this is not a hierarchical arrangement; both sets of behaviors depend on each other, and have the power to influence each other's execution. This also conserves the spirit of BBS systems, which do not tend to employ top-down hierarchies.

Experimental validation

To validate the proposed concepts, we implemented them on a physical mobile robot given an object delivery task. The delivery task consists of a mobile robot in an enclosed environment divided into two sections, separated by a swinging door. The robot must find a box, which may be in either section, and push it to the delivery point.

The BBS controller must accommodate various initial conditions: the robot may be in the same section as either the box and/or the delivery point, and the box may or may not be in the same section as the delivery point. Note that this is not a large state space, which is why it lends itself to BBS solutions, but it is sufficient versatile that it would require several different plans if pursued in a deliberative fashion. Our approach uses two networks which, together, account for all possibilities, and, as any BBS, adapts to uncertainty and changes that may occur (i.e., the robot or the box or both can be moved at any point).

The experiments were performed on a Pioneer 2-DX mobile robot, equipped with two rings of sonars (8 front and 8 rear), a pan-tilt-zoom camera, a PC104 with an Intel2 processor at 233MHz and 32Mb of memory. All behaviors are sensor-based; the robot uses its camera to detect the delivery point and the box, and determine the state of the door (open/closed).

The controller has been implemented using AYLLU (Werger 2000), an extension of the C language for development of distributed control systems for groups of mobile robots.

Behavior Networks for Delivery

The solution for the delivery task was constructed by hand from the following repertoire of behaviors:

- **Localize** - the robot wanders around in order to localize itself with respect to Home. Achieves *Location* $\neq$ *Unknown.*
- **GetBox** - the robot wanders in search of the box. Achieves *HaveBox = True* or signals *Timeout* in case the box cannot be found within a predetermined period of time in the current room.
- **GoTo(Door)** - the robot goes to the door. Achieves *AtPlace(Door) = True.*
- **OpenDoor** - the robot opens the closed door. Achieves *DoorOpened = True* and *HaveBox = False* (since the robot cannot carry anything in order to be able to open the door). Dealing with this type of conflicting goals is described in the next section.
- **GoThroughDoor** - the robot goes through the door to the next room. Achieves *SideRoom(RoomX) = True*; where Room X is a parameter bound to Home.
- **GoTo(Home)** - the robot goes to its Home location. Achieves *AtPlace(Home) = True.*

Two different task plans have been developed by hand for the delivery task and have been translated into behavior networks (Figure 3) that use the behavior set above. The robot automatically switches between the networks at run-time, according to predefined changes in the robot's internal state. This is the only built-in specific information in our system; it could have been avoided if external cues that could be sensed directly were available. In that case, we could have directly informed the robot when a network switch should occur.

It is important to note that since our networks rely on real behaviors which can handle a variety of initial conditions, we do not need to have a "plan" and thus a behavior network for each initial condition. Our solution makes use of only two alternate "plans" for the four possible initial conditions. This, of course, is not the only solution for the task, but we have chosen it because it captures the important aspects of the representation that we want to validate: 1) reuse of behaviors for different (sub)tasks without behavior redesign and 2) recompilation and dynamic switching between behavior networks.

The robot begins with the localization behavior (the only one for which all the execution conditions are met at that point) in order to determine in which room it is. Its goal of knowing the current location is a task precondition for all other behaviors (as can be seen by the network links from *Localize* to all other behaviors). Once localized, the robot starts looking for the box. If it finds it within a predetermined time, it continues to execute the current behavior network. If it fails to find the box, timeout is signaled, and the robot switches to another "plan", represented by the second behavior network. The alternate solution is to go to the other room, and look for the box there. The same *GoToDoor*, *OpenDoor*, *GoThroughDoor* behaviors are used in both networks. Even if the task specific conditions that they are testing are different, no change has to be done to the behaviors themselves; they continue to run as before, only they check the *Effects* outputs of a different set of behaviors. For example, the second network need not test the status of *GetBox* in order to go to the door and through it, as it would if the box had been found. At the completition of the alternate "plan" represented by the second behavior network, the robot switches again to the initial network and starts looking for the box in the room it is now in.

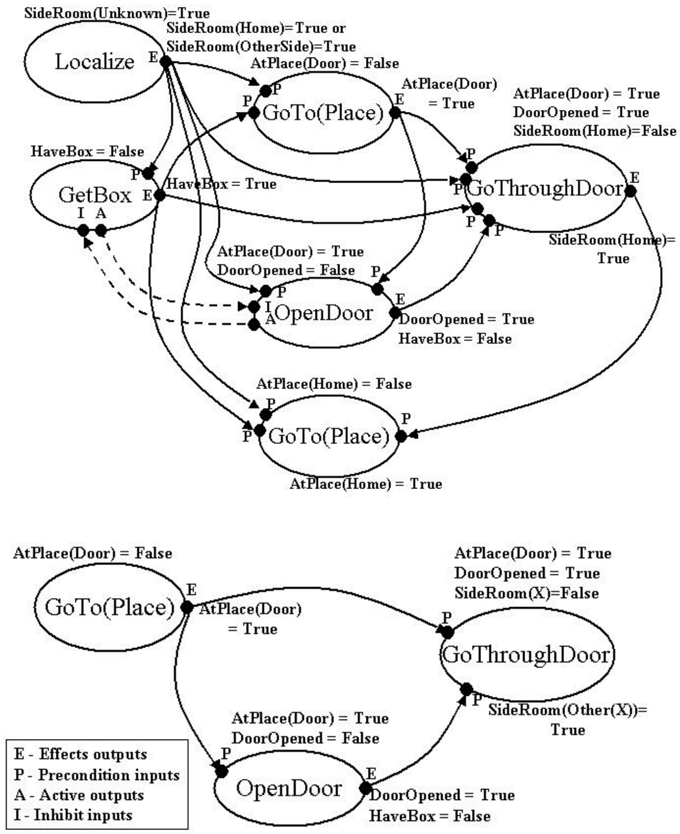

Figure 3: Structure of the behavior networks for the delivery task

Each of the two behavior networks that we employed represents a solution to a different problem by itself: the first one is a solution for the delivery problem when both the robot and the box are in the same room and the second one is a solution for the task of going from one room to another. They both rely on the same set of behaviors and the specifics of each tasks requires the behaviors to check different activation conditions in each case. However, due to the fact that those preconditions are embedded in the network topology (the Effects-Precondition links), the behaviors can be reused without changes for different tasks and the tasks can be switched dynamically by simply rearranging those links.

Competitive behaviors

In the delivery task, behavior competition (first class of behavior competition we described in the Behavior Network Construction and Execution Section) arises between the *GetBox* and *OpenDoor* behaviors. While the former drives the robot to the box if it does not have it yet, the latter requires pushing the box aside in order to open the door. After getting the box, the *GetBox* behavior is no longer active and no longer inhibits *OpenDoor*. When *OpenDoor* becomes active, it inhibits *GetBox* until the door is opened. At that point it deactivates and in turns stops inhibiting *GetBox*, allowing the robot to again find the box and take it home through the opened door.

Results

To demonstrate the validity of our representation, and the ability to dynamically switch between behavior networks, we tested the delivery task from all four different initial conditions. For each of them we ran the robot four times, once with the door closed and the rest with it open. We found that irrespective of the initial conditions, the robot adapted itself to the state of the environment, activated the correct behavior network for that state, and executed its actions accordingly.

As another validation of the generality and a demonstration of behavior reuse that the abstract representation provides to behaviors, we have run an experiment with the opposite task of cleaning (i.e., taking the box out from the room where the delivery point is), using the same set of behaviors and a slightly changed version of the behavior networks. For the first network we no longer need the *GoTo(Home)* behavior; the goal of *GoThroughDoor* becomes *Other(Home)*. For the second network the goal of *GoThroughDoor* becomes *SideRoom(Home)=True*. Irrespective of the two initial positions of the robot and the box that we have tested (the first with both the robot and the box in the section with the delivery point, the second with the box in the delivery section and the robot in the other), the robot was able to reliably push the box out of the delivery section.

Extensions and Continuing Work

The immediate extension of this work that we are already pursuing is to generalize it to other tasks, to the point where our high-level behavior representation can be employed with a generic behavior library and the process of BBS construction can be largely automated, at least within the class of tasks that are satisfied by the given behavior set. We have briefly discussed the principles of our algorithm for automatic network generation and the important issues that have to be solved along with it.

Another extension of this work, and one of its underlying motivations, is to address human-robot interaction. In (Nicolescu & Matarić 2000) we have demonstrated how the use of abstract behaviors allows us to employ simple communication mechanisms for interaction, enabling the human to help the robot, and the robot to benefit from the human and also learn about cooperation. Without the use of abstract behaviors, such interaction would be much more complex, as the human would need to have access to the inner workings of the behaviors.

As the next goal, we plan to use the abstract behavior representation to enable multiple robots to share their acquired knowledge and experiences of cooperation, in order to facilitate further cooperation and learning at the level of the group. Here again, by employing high-level behavior abstractions, we can establish smooth interaction between robots using different types of behaviors, sensors, effectors, and having different goals.

Related Work

The abstract behavior representation proposed in our paper combines the advantages of deliberative, STRIPS-like architectures (Fikes & Nilsson 1971) that operate at a high level of abstraction, and those of the BBS' capability to operate in uncertain, unstructured, dynamically changing environments.

The most common approach researchers have used in order to bridge the gap between these architectures is the use of the hybrid systems. Such architectures, also called *three-layer systems* employ a symbolic deliberative layer and a reactive layer, with a middle layer that resolves the difference in the time-scales and representations used by the other two (Gat 1998). Agre & Chapman (1990) used a planner to give advice to the reactive control system, which could choose to use or ignore the advice. Arkin & Balch (1997) proposed a hybrid strategy that integrated a symbolic, deliberative level with a reactive (schema-based) controller, as a selection tool for the behavioral composition and parameters used during execution. In the above examples, since the behaviors themselves did not contain any type of representation, in order to perform a more complex task, behaviors had to be activated from a higher level, which runs at a slower time scale and uses a different representation. In contrast, our architecture does not alter the nature of behavior-based systems and allows complex controllers to be specified in terms of real behaviors having the same representation and time scale.

Different hybrid architectures, relying on the Discrete Event Systems (DES) theory, have been proposed by Kosecká & Bogoni (1994) and by Huber & Grupen (1997). These address the problem of deriving control policies from a given set of underlying behaviors by resorting to a mechanism for "supervisor synthesis", which combines the state machines of elementary behaviors into a new single state machine representing the composite behavior. The controller thus obtained also contains the constraints required by the system in order to properly execute a given task. Although these behaviors are also reused for any new controller generation, the result does not preserve the representation of the elementary behaviors that composed it, making it harder to understand or debug the resulting controller and also to generate more complex solutions, due to the increase of the number of states and task complexity. Our behavior networks work at a more abstract level, preserve the behaviors that compose it, use the representation of their goals to naturally sequence behaviors and solve the behavior conflicts with simple inhibition links between the behaviors.

Important research has been done in the area of systems with "circuit semantics" in order to ground representations in dependency networks driven by sensors. Kaelbling & Rosenschein (1990) propose the *situated automata* model which captures the fundamental relationship between an agent and its environment. Their implementation, however, does not allow generalization and reuse of the compiled high level circuitry and the representation is only limited to propositional logic. Nilsson (1994) introduced the notion of T-R programs which, like our networks of abstract behaviors, provide behavior-based systems the capability of being used for complex tasks that involve temporal sequences. However, T-R programs themselves do not have any representation of the behavior goals: the designer uses the approximate information about what conditions may be true at the end of the behavior execution in order to design the condition/action rules and their hierarchy. This makes the representation somehow unnatural for use in conjunction with an automatic system for generating T-R programs for a given goal, unless, as the author mentions, incremental modification of the constructed trees is made until they are more and more matched to the agent's environment. Horswill (1997) presents an extension of parallel reactive architectures, using "role-passing" in order to implement a subset of modal logic with circuit semantics. The architecture allows dynamic binding of a fixed set of indexical terms and supports universal quantification over those names. Also, by allowing complex predicates to be tracked by a dependency network, the system is able to represent its own attentional state over the binded terms. The system is very related to ours in that both use a limited version of binding ("role-passing" and behavior parameters) to ground binary predicate representations in sensory data within a behavior-based architecture. However, parallel reactive architectures are still limited by the dynamical growth of their dependency networks.

A special approach to robot programming has been done by Lyons & Arbib (1989). Their *Robot Schemas*, formally defined using the *port automaton model*, allow for constructing nested robot task representations. Our network topology, however, is different in that it embeds the task-specific dependencies in behavior links that can be dynamically changed, rather than using hand-coded preconditions-postconditions specifications.

An early example of embedding representation into BBS was done by Matarić (1992). The representation was also constructed from behaviors, and was used exclusively for mapping and path planning. While the approach successfully integrates the deliberative component into the behavior representation, and is thus related to our work, it is limited only to the navigation task, while our representations are meant to be task-independent.

Maes (1990) and Maes (1989) describe an action selection mechanism for a situated agent, based on spreading activation within a network created dynamically from a given behavior repertoire. This approach is very related to ours, since network nodes are reused for different goals that can be changed at run-time. A key difference is that the network nodes, at least in the demonstrated examples, consist of STRIPS-like high-level operators much more abstract than those we employ. Furthermore, the work assumes that the network and its topology are provided by the designer, while we describe a process by which the networks (typically much less complex, due to the higher level behavioral representation) can be automatically generated.

Conclusions

We have addressed two limitations of behavior-based systems: the lack of abstract representation, which makes them unnatural for complex problems containing temporal sequences, and the lack of generality, which requires system redesign from one task to another, even if the underlying behavior remain unchanged. We have shown how to adapt behaviors to an abstract representation and employ those in the construction and use of behavior networks. We validated the proposed ideas on a mobile robot delivery task, which featured multiple initial conditions and behavior competition. The system was able to automatically switch between two different behavior networks, both making use of the same underlying behavior set. In our continuing work we plan to further generalize this approach toward much more automated behavior-based system design and behavior re-use.

References

Agre, P. A. & Chapman, D. (1990), 'What are plans for?', *Journal for Robotics and Autonomous Systems* **6**, 17–34.

Arkin, R. C. (1998), *Behavior-Based Robotics*, MIT Press, CA.

Arkin, R. C. & Balch, T. (1997), 'AuRA: Principles and Practice in Review', *Journal of Experimental and Theoretical AI* **2-3**, 175–189.

Chapman, D. (1987), 'Planning for Conjunctive Goals', *Aritifical Intelligence* **32**, 333–377.

Fikes, R. E. & Nilsson, N. J. (1971), 'STRIPS: A new approach to the application of theorem proving to problem solving', *Artificial Intelligence* **2**, 189–208.

Gat, E. (1998), On Three-Layer Architectures, *in* D. Kortenkamp, R. P. Bonnasso & R. Murphy, eds, 'Artificial Intelligence and Mobile Robotics', AAAI Press.

Horswill, I. (1997), Real-time control of attention and behavior in a logical framework, *in* W. L. Johnson & B. Hayes-Roth, eds, 'Proceedings of the First International Conference on Autonomous Agents (Agents'97)', ACM Press, New York, pp. 130–137.

Huber, M. & Grupen, R. (1997), 'A feedback control structure for on-line learning tasks', *Robotics and autonomous systems* **22**(3–4), 303–315.

Kaelbling, L. P. & Rosenschein, S. J. (1990), 'Action and planning in embedded agents', *Robotics and Autonomous Systems(1&2), June 1990* **6**, 35–48.

Kosecká, J. & Bogoni, L. (1994), Application of discrete events systems for modeling and controlling robotic agents, *in* E. Straub & R. S. Sipple, eds, 'Proceedings of the International Conference on Robotics and Automation', Vol. 3, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 2557–2562.

Lyons, D. M. & Arbib, M. A. (1989), 'A formal model of computation for sensory-based robotics', *IEEE Transactions on Robotics and Automation* **5**(3), 280–293.

Maes, P. (1989), 'How to do the right thing', *A.I. Memo No 1180*.

Maes, P. (1990), 'Situated Agents Can Have Goals', *Journal for Robotics and Autonomous Systems* **6**(3), 49–70.

Matarić, M. J. (1992), 'Integration of Representation Into Goal-Driven Behavior-Based Robots', *IEEE Transactions on Robotics and Automation* **8**(3), 304–312.

Matarić, M. J. (1997), 'Behavior-Based Control: Examples from Navigaton, Learning, and Group Behavior', *Journal of Experimental and Theoretical Artificial Intelligence* **9**(2–3), 323–336.

Nicolescu, M. N. & Matarić, M. J. (2000), Learning cooperation from human-robot interaction, *in* 'Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems (DARS), Knoxville, TN, Oct 4-6, 2000'.

Nilsson, N. J. (1994), 'Teleo-reactive programs for agent control', *Journal of Artificial Intelligence Research* (1), 139–158.

Pirjanian, P. (1999), Behavior Coordination Mechanisms - State-of-the-art, Tech Report IRIS-99-375, Institute for Robotics and Intelligent Systems, University of Southern California, Los Angeles, California.

Werger, B. B. (2000), Ayllu: Distributed Port-Arbitrated Behavior-Based Control, *in* 'Proceedings of the 5th International Symposium on Distributed Autonomous Robotic Systems (DARS), Knoxville, TN, Oct 4-6, 2000'.