In *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems*
Bologna, ITALY, July 15-19, 2002

A Hierarchical Architecture for Behavior-Based Robots

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

ABSTRACT

Behavior-based systems (BBS) have been effective in a variety of applications, but due to their limited use of representation they have not been applied much to more complex problems, such as ones involving temporal sequences, or hierarchical task representations. This paper presents an approach to implementing these AI-level concepts into BBS, without compromising BBS' key properties. We describe a *Hierarchical Abstract Behavior Architecture* that allows for the representation and execution of complex, sequential, hierarchically structured tasks within a behavior-based framework. The architecture, obtained by introducing the notion of *abstract behaviors* into BBS, also enables reusability of behaviors across different tasks. The basis for task representation is the *behavior network* construct which encodes complex, hierarchical plan-like strategies. The approach is validated in experiments on a Pioneer 2DX mobile robot.

1. INTRODUCTION

Behavior-based control [2, 13] has become one of the most popular approaches to embedded system control both in research and in practical applications. Behavior-based systems 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 present an approach for embedding representations into BBS without compromising their key philosophy of non-hybrid representation and real-time execution. Toward this end we developed a *Hierarchical Behavior-Based Architecture* that addresses two limitations of BBS, both having to do with the use of representation.

The first limitation is the fact that behaviors lack the abstract (symbolic) representation that would allow them to be employed at a high level, like operators in a plan. Behaviors are typically invoked by built-in reactive conditions, and as a consequence, BBS are typically unnatural for, and thus rarely applied to complex problems that contain temporal sequences or hierarchical structures. Since we seek a method that allows for encoding general (possibly hierarchical) tasks that would require the sequential activation of the robot's behaviors, we need a mechanism that would allow first the representation and then the execution of such sequences/hierarchies.

The second, and related, limitation is that the vast majority of BBS are still designed by hand for a single task: the standard behavior architecture prevents the automatic reusability of behaviors across different tasks and thus the automatic generation of BBS. Although behaviors themselves, once refined, are usually reused by designers and gradually accumulate into behavior libraries, 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. Our other work addresses techniques for automatic generation of behavior networks [14], but that is outside of the scope of this paper, which focuses on a BBS architecture that lends itself to these otherwise typically AI methods.

Attempts to solve these issues have resulted either in hybrid architectures [9], or in behavior-based architectures that only partly address the above problems. We propose a representation that implements these AI concepts into a BBS without compromising the key principles of behavior-based systems. We present a detailed discussion of the differences between existing architectures and ours in Section 5.

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 employ 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 Section 5).

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 reminder of the paper we first describe the process of adapting behaviors for representation, then introduce the notion of *abstract behaviors* and the *behavior network* construct that uses them to represent general strategies and plans. We describe how

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AAMAS'02, July 15-19, 2002, Bologna, Italy.
Copyright 2002 ACM 1-58113-480-0/02/0007 ...$5.00.

these plans can be defined, and finally, we validate them in real robot experiments. We end the paper with a review of related work, directions for future work and conclusions.

2. 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 behavior's activation status: 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 from 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 were 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 [8]. 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.

Abstract and primitive behaviors can both be quite complex, just as they are within any system embedded in an environment. The

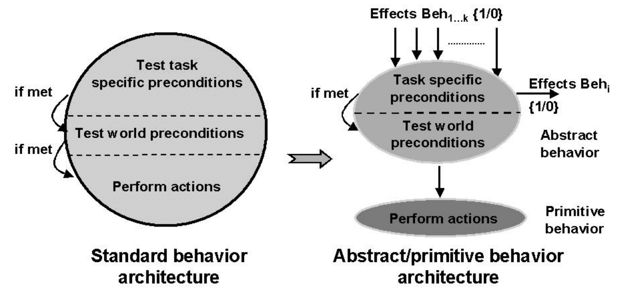

Figure 1: Adaptation of standard behaviors for abstract representations

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 task 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.

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

3. BEHAVIOR REPRESENTATION

3.1 Abstract Behaviors

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 with 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 well suited for high-level, complex tasks, is that they are parameterizable. The behavior goals are abstracted environmental states, which can also be represented in a "predicate-like" form on the behavior parameters. For example, a **Tracking** behavior's goal could be **DistanceToTarget** = *GoalDistance*, where the distance to the target is obtained from the sensory input and the *GoalDistance* is the behavior's parameter. It is important to notice that the effects of behaviors are continuously computed from the

robot's sensors and not high-level symbols that are not grounded in direct 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.

As with operators in a plan, behaviors can undo each other's actions while trying to achieve their own goals [7]. In BBS, such undesirable competition is typically handled either by mutually-exclusive behavior execution conditions, or by the behavior coordination mechanism [16]. 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.

Structurally, behaviors are composed of a set of processes, running continuously and concurrently with other behaviors, and an interface of input and output ports with which they can communicate with other behaviors. The implementation presented here utilizes the Port-Arbitrated Behavior paradigm presented by [19].

An *Abstract Behavior* has the following Input Ports (Figure 2):

- *UseBehavior* (binary input): signals if the behavior is used in the current network controller. The behavior is *enabled* if the port has a value of 1, and *disabled* otherwise. This input is important to the hierarchical representation, as described in Section 3.4.
- *ActivLevel*: sums the activation messages received from other behaviors; its value represents the behavior's activation level.
- *Inhibit* (binary input): a value of 1 signals that the behavior is inhibited by another behavior, a value of 0 signals that it is not.
- *Sensory Input*: a set of inputs from the environment, required to continuously compute the status of the behavior's goals.
- $Preconditions_{1\ldots k}$: inputs from predecessor behaviors, whose execution influence the activation of the current behavior.
- *Continue* (binary input): coming from the corresponding primitive behavior(s) (discussed below).

The *Abstract behavior* Output Ports are:

- *Active* (binary output): activates/deactivates the corresponding primitive behavior(s).
- *Effects* (binary output): signals the current status of the behavior's postconditions as computed from the sensory data.

A *disabled* or *inhibited* behavior does not perform any type of computation for the task. If *enabled* and *non-inhibited*, a behavior runs at a predefined rate at which it continuously checks or sends its inputs and outputs. However, only if *active*, the behavior will actually be allowed to send its action commands to the robot's actuator. 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.).

The abstract behavior activation mechanism is presented in Section 3.3, which also presents the methods for encoding and executing tasks in the form of behavior networks.

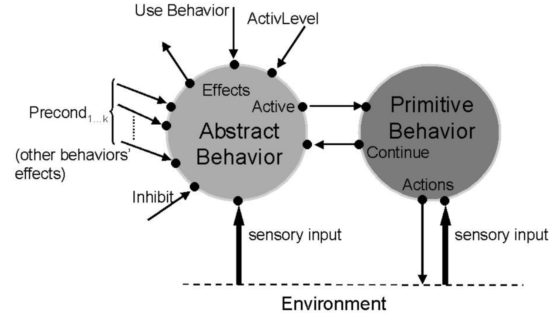

Figure 2: Input/output structure of abstract and primitive behavior.

3.2 Primitive Behaviors

Primitive behaviors (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.

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 yet finished 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 completion. In all other situations, the abstract behavior can stop sending its activation at any time, according to its current conditions.

3.3 Behavior Networks

3.3.1 Components and Structure

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 completion 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 re-executed). Also, by introducing the *Network Abstract Behavior* construct (Section 3.4) we allow for hierarchical task representations.

Second, by encoding the task-specific sequences into the network links, we allow behaviors to be reused for different tasks. In addition, since behaviors' goals could also be represented in a "predicate-like" form, they become suitable for use with a general purpose planner, similarly to classical planning operators, in order to obtain a solution for a given task. Our behavior networks, since they rely on real behaviors, also have the advantage that they could handle a variety of initial conditions within a single task representation, in contrast to typical plan-representations which are different

for distinct initial conditions.

We distinguish between three types of *sequential preconditions*, which determine the activation of behaviors during the behavior network execution.

- **Permanent preconditions**: preconditions that must be met during the entire execution of the behavior. A change from met to not met in the state of these preconditions automatically deactivates the behavior. These preconditions enable the representation of sequences of the following type: *the effects of some actions must be permanently true during the execution of this behavior*.

- **Enabling preconditions**: preconditions that must be met immediately before the activation of a behavior. Their state can change during the behavior execution, without influencing the activation of the behavior. These preconditions enable the representation of sequences of the following type: *the achievement of some effects is sufficient to trigger the execution of this behavior*.

- **Ordering constraints**: preconditions that must have been met at some point before the behavior is activated. They enable the representation of sequences of the following type: *some actions must have been executed before this behavior can be executed.*

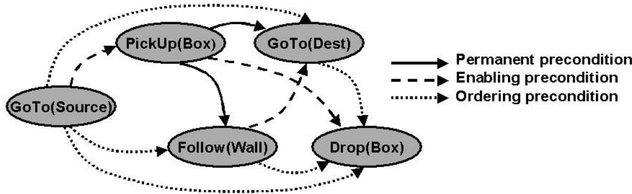

Figure 3: Example of a behavior network

From the perspective of a behavior whose goals are **Permanent preconditions** or **Enabling preconditions** for other behaviors, these goals are what the planning literature calls goals of *maintenance* and of *achievement*, respectively [17]. In a network, a behavior can have any combination of the above preconditions. The goals of a given behavior can be of maintenance for some successor behaviors and of achievement for others. Thus, since in our architecture there is no unique and consistent way of describing the conditions representing a behavior's goals, we distinguish them by the role they play as preconditions for the successor behaviors. Figure 3 shows an example behavior network and the three types of precondition-postcondition links.

3.3.2 Behavior Network Execution

Behavior networks allow for two different modes of execution within the same representation, depending on the constraints on various parts of the task:

- *Sequential execution*, for the task segments containing temporal ordering constraints;

- *Opportunistic execution*, for the task segments for which the order of execution does not matter.

Sequentiality is enforced by the existence of precondition - postcondition dependencies between behaviors whose execution needs to be ordered. Opportunistic execution is achieved by not placing temporal dependencies between the behaviors which do not require a particular ordering. The ability to encode both these modes of execution within the same behavior network increases the expressive power of the architecture, through the embedding of multiple paths of execution within the same representation.

In a network requiring only *sequential execution*, since all behaviors are connected with ordering constraints, there can only be a single behavior that can be *active* at a given time. By introducing the *opportunistic* mode of execution, we allow multiple behaviors to be "suitable" for activation at the same time, if their own activation conditions are met. This raises the problem of concurrent access to the robot's actuators. To deal with this issue we choose the solution of *locking* the actuators that are used by a behavior for the entire duration while it is active. The method implements implicit inhibition of behaviors at the actuator level. This provides a natural way of preventing multiple behaviors to have access to the same actuators, while still enabling the simultaneous execution of behaviors that control sets of actuators that are disjunct.

All the behaviors that are used in a network (i.e., have their *Use-Behavior* port set) are continuously *running* (i.e., performing the computation described below), but only the behaviors that are *active* are sending commands to the actuators at a given time. A default **Init** behavior initiates the network links and detects the completion of the task.

Similarly to [11], we employ a continuous mechanism of activation spreading, from the behaviors that achieve the final goal to their predecessors (and so on), as follows: each behavior has an *Activation level* that represents the number of successor behaviors in the network that require the achievement of its postconditions. Any behavior with activation level greater than zero sends activation messages to all predecessor behaviors that do not have (or have not yet had) their postconditions met. The activation level is set to zero after each execution step, so it can be properly re-evaluated at each time, in order to respond to any environmental changes that might have occurred.

The activation spreading mechanism works together with precondition checking to determine whether a behavior should be active, and thus able to execute its actions. A behavior is activated iff:

(It is used in the current controller) AND
(It is not inhibited) AND
(Its controlled actuators are not locked) AND
(The *Activation level* $!= 0$) AND
(All **ordering constraints** $= TRUE$) AND
(All **permanent preconditions** $= TRUE$) AND
((All **enabling preconditions** $= TRUE$) OR
(the behavior was active in the previous step))

In the current implementation, checking the precondition status is performed serially, but the process could also be implemented in parallel hardware.

3.4 Hierarchical Behavior Networks

As robot tasks become more complex and begin to rely on previously developed skills, it is useful to have a hierarchical task representation, which can encapsulate the complex dependencies.

The behavior network representation described so far allows only for *flat representations*, in which all the components are *abstract behaviors*. While the architecture is expressive and flexible, it does not have the modularity needed when new, more complex tasks would have to be created from already existing ones. The best solution would be to specify the new task using abstractions of these

existing modules, rather than combining their underlying behaviors into a larger, flat network. In this way, only the precondition-postcondition dependencies at the higher-level (between the two sub-networks) would have to be specified, reducing the connectivity of the network.

We enable this higher-level of representation by introducing the notion of a *Network Abstract Behavior* (NAB), which abstracts away an entire behavior network into a single component. This allows the construction of hierarchical representations of robot tasks, whose components can be either *Abstract Behaviors* (ABs) or NABs, which can be further decomposed into lower level representations. An example of a generic hierarchical network representation is presented in Figure 4.

Functionally, a NAB is equivalent to a regular abstract behavior, in that it performs the same computation and plays the same role in the network. The postconditions of a NAB will be true when the execution of the subnetwork it represents is finished. The only difference between a NAB and an AB is in the connection of their *Active* output. For an abstract behavior, the *Active* output is connected to the *Active* input of the corresponding primitive behavior(s). For a NAB, the *Active* output is connected to the *UseBehavior* input of the corresponding component ABs or NABs. Thus, when a NAB is not active, all behaviors (ABs or NABs) which are a part of the subnetwork it represents are disabled, and therefore can be regarded as nonexistent for the task. When a NAB becomes active, all its underlying behaviors are enabled, and the subnetwork becomes the current "network" that is being executed. When the execution of the subnetwork finishes, the NAB signals to the successor behaviors the achievement of its goals, just as regular ABs do, and execution continues at the level of the network containing the NAB.

Formally, a behavior network is described as follows:

NETWORK-DESCRIPTION =
$<$ **Number of components (N),**
$\{$**Component-Description**$\}_N$**,**
Topology-Description $>$

where,

Component-Description =
$<$ **AB-Description | ABN-Description** $>$

AB-Description =
$<$ **Component-ID,**
BehaviorID, Number of Parameters (P),
$\{$**Parameter Name, Parameter Value**$\}_P$ $>$

NAB-Description =
$<$ **Component-ID,**
NETWORK-DESCRIPTION $>$

Topology-Description =
$<$ **Number of Links (L),**
$\{$**FromComp-ID, ToComp-ID, Link-Type**$\}_L$ $>$

Link-Type =
$<$ **Ordering | Enabling | Permanent** $>$

This formalism describes a behavior network by the number N of its components (ABs or NABs), their descriptions, and the topological links between them. The **Component-ID** is a unique identifier of the component within the network and the **FromComp-ID** and **ToComp-ID** are the IDs of the start and respectively end-points of a network link.

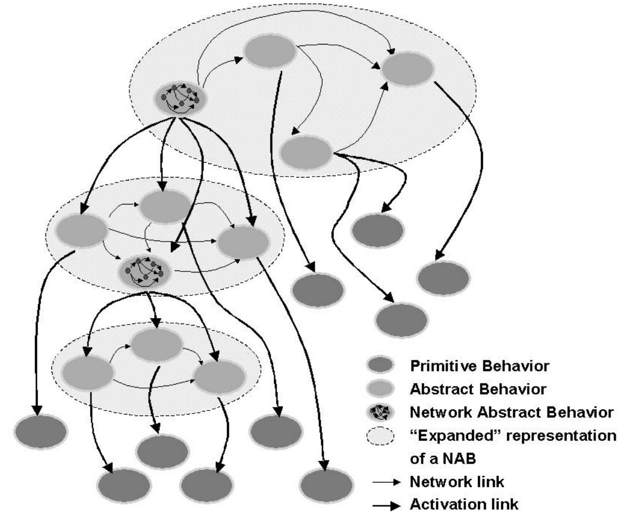

Figure 4: A generic hierarchical task representation

4. EXPERIMENTAL RESULTS

We implemented and tested our concepts on a Pioneer 2-DX mobile robot, equipped with two rings of sonars (8 front and 8 rear), a SICK laser range-finder, a pan-tilt-zoom color camera, a gripper, and on-board computation on a PC104 stack. We performed the experiments in a 5.4m x 6.6m arena. The robot was programmed using AYLLU [19], an extension of C for development of distributed control systems for mobile robots. The robot has a behavior set that allows it to track colored targets, pick up, and drop objects:

- **PickUp(ColorOfObject)** - the robot picks up an object of the color *ColorOfObject*. Achieves *HaveObject = TRUE*.
- **Drop** - the robot drops what it has between the grippers. Achieves *HaveObject = FALSE*.
- **Track(ColorOfTarget, GoalAngle, GoalDistance)** - the robot tracks a target of the color *ColorOfTarget* until it gets at *GoalDistance* and *GoalAngle* to the target. Achieves *DistToTarget = GoalDistance AND AngleToTarget = GoalAngle*.

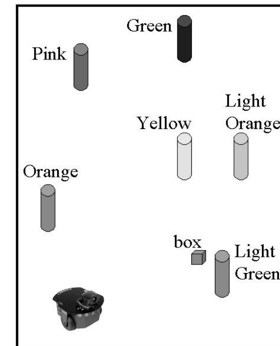

Figure 5: The environmental setup

The goal of the validation experiments is to demonstrate the key features of the presented architecture: hierarchical task representation, behavior reusability, and the ability for both sequential and opportunistic execution.

Toward this end, we considered a task consisting of sequencing of two subtasks: an **Object transport** task and a **Visit targets**

task (Figure 6). The setup for this experiment is presented in Figure 5. The **Object transport** task requires the sequential execution of its steps: go to the light green target, pick up the orange box, go through the gate formed by the yellow and light orange targets, go to the green target and drop the box there. As the figure shows, **GoThroughGate** itself has a subtask representation. The **Visit targets** task does not enforce the ordering of the target visits, thus allowing the robot to perform the task according with the particularities of the environment (i.e., visit the Pink, Light-Green, Yellow and Orange targets in the order in which they are encountered).

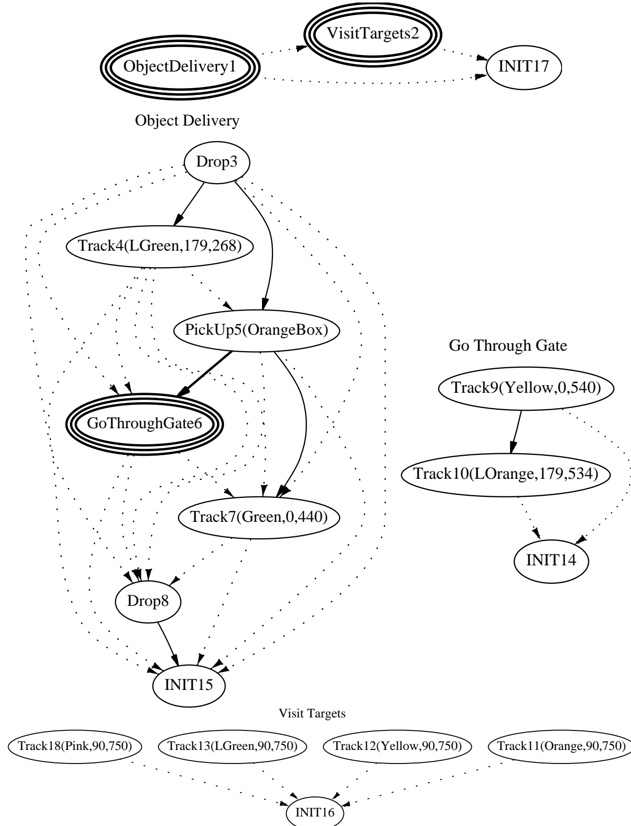

Figure 6: The hierarchical network representation. The subtasks (NABs) have 3-line borders, and the ABs have one-line borders. The numbers in the behaviors' names are their unique IDs.

Table 1: Order of target visits

Trial 1	Orange Pink Light-Green Yellow
Trial 2	Pink Yellow Orange Light-Green
Trial 3	Light-Green Yellow Orange Pink
Trial 4	Yellow Light-Green Orange Pink
Trial 5	Yellow Orange Pink Light-Green

We performed 5 experiments; in all five, the robot correctly executed the task. The order in which the robot visited the targets during the **Visit targets** subtask is shown in Table 1. Since the robot's paths are different from one experiment to another, due to limited sensing and uncertainty in searching for the colored targets, the robot opportunistically visited the targets as it encountered them.

We thus show that both *sequential* and *opportunistic* execution can be enforced and respectively allowed in a unique task representation, within a behavior-based framework.

4.1 Discussion

Several aspects of the experiments presented above show the advantage of the hierarchical representation. First is the ability to encode tasks of increasing levels of abstraction, which facilitates the modular representation of higher-complexity tasks in the behavior-based framework. Second is that by "abstracting" already known tasks into NABs, the complexity (connectivity) of the networks that might include those NABs as subtasks is greatly reduced. The abstraction eliminates unnecessary network links that would have to be specified **to** and **from** all the behaviors of a subtask in order to ensure proper execution and sequencing. For the experiment presented above, the number of network links would be increased from 31 to 60, from a hierarchical to a flat representation.

In the task representation presented above, based on a given set of behaviors, multiple instantiations of the same behavior are used within the same NAB or in separate NABs, without customization or redesign, although in each case they have different activation conditions. Due to the fact that those preconditions are embedded in the network topology, the behaviors can be reused without changes in circumstances requiring different activation conditions.

5. RELATED WORK

By augmenting the behaviors with representations of their goals, we take advantage of both the ability of the deliberative, STRIPS-like architectures to operate at high-level of abstractions, and the robustness of BBS. The common approach to bridging the gap between these architectures is the use of the hybrid (or three-layer) systems [9], [1], [3], [6], which need a middle layer to interface between the different representations and time-scales between the physical and the abstract levels. In these examples, since the behaviors themselves do not contain any type of representation, in order to perform a more complex task, behaviors have 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.

In [6] for example, the *Sequencer* (which activates the robot's *skills* to perform a given the task) relies on *event monitors* (sent from the skill manager) to detect when certain world state has been achieved. While this approach requires the use of *wait-for* statements to block the task until event messages are received and posted into a RAP memory, within a behavior network the transition between task steps flows naturally since the behaviors are able to continuously detect the relevant environmental conditions.

An early example of embedding representation into BBS was done by [12]. The representation was also constructed from behaviors, and was used exclusively for mapping and path planning. While the approach successfully integrates deliberative capabilities into a BBS, it is limited to the navigation task, while our representations are meant to be task-independent and could embed any general behaviors representing the robot's capabilities: in our case, both navigation and object manipulation.

An approach to robot programming related to ours has been done by [10]. They introduce the notion of *Robot Schemas*, formally defined using the *port automaton model* [18], and *Assemblage Schemas* to construct nested robot task representations. A key difference from their work is that in our approach behaviors may interact by actually enabling/disabling or activating/inhibiting each other. As presented in the description of the NABs' execution, this approach also reduces the computational effort, since only the behaviors within the currently active subnetwork would be running, the

rest of them being disabled.

Hierarchical architectures for behavior control have also been developed for agents embedded in virtual environments. [5] describes a *Parameterized Action Representation (PAR)*, to hierarchically encode the actions of a virtual human agent. The *Hierarchical Agent Control Architecture (HAC)* presented in [4] uses three hierarchies: for action, for sensors, and for context. The hierarchy for structuring the sensory information into increasing levels of abstraction is similar to our goal representation for the *abstract behaviors*, but it is not linked with the behaviors whose goals it represents. Both the above methods also maintain models of the environment and update these models as a result of the agent's actions, an approach which is generally not practical in dynamic real-world domains. Another difference is that the knowledge about the ordering of the steps in a task is maintained at a higher-level responsible with the activation of high-level actions or PARs [4]; we encode this task-specific ordering information into behavior links which naturally make the transition from one behavior's execution to another.

6. FUTURE WORK

Our architecture, in its non-hierarchical form, has been successfully used for learning task representations from demonstrations of both human and robot teachers [14], and also for robot-human communication and interaction [15]. An immediate extension of this work is to use the architecture for learning hierarchical task representations, by abstracting already learned, flat representations and/or directly from demonstration, by using a minimum set of guiding symbols that could signal parts of the task that can be encapsulated together.

At the level of the expressive power of the representation, we are interested in including the possibility for alternate (disjunct) activation conditions that would enable behaviors to be activated by a richer set of conditions.

7. CONCLUSION

This paper presented a Hierarchical Abstract Behavior Representation, which extends standard behavior-based architectures with typically AI-level concepts. Toward this end we addressed several limitations of BBS: 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 a task to another. The architecture also allows for hierarchical task decomposition and both *sequential* and *opportunistic* modes of execution. The approach was validated on experiments with a Pioneer 2DX mobile robot.

Acknowledgments

This work is supported by DARPA Grant DABT63-99-1-0015 under the Mobile Autonomous Robot Software (MARS) program and by the ONR Defense University Research Instrumentation Program Grant N00014-00-1-0638.

8. REFERENCES

[1] P. A. Agre and D. Chapman. What are plans for? *Journal of robotics and autonomous systems(1&2), June 1990*, 6:17–34, 1990.

[2] R. C. Arkin. *Behavior-Based Robotics*. MIT Press, CA, 1998.

[3] R. C. Arkin and T. Balch. Aura: Principles and practice in review. *Journal of Experimental and Theoretical AI*, 2-3:175–189, Apr-Sep 1997.

[4] M. S. Atkin, G. W. King, D. L. Westbrook, B. Heeringa, A. Hannon, and P. Cohen. Spt: Hierarchical agent control: a framework for defining agent behavior. In *Proc., Intl. Conf. on Autonomous Agents*, pages 425–432, May 2001.

[5] R. Bindiganavale, W. Schuler, J. M. Allbeck, N. I. Badler, A. K. Joshi, and M. Palmer. Dynamically altering agent behaviors using natural language instructions. In *Proc., Intl. Conf. on Autonomous Agents*, pages 293–300, June 2000.

[6] R. P. Bonasso, R. J. Firby, E. Gat, D. K. D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive systems. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2–3):237–256, 1997.

[7] D. Chapman. Planning for conjunctive goals. *Aritifical Intelligence*, 32:333–377, 1987.

[8] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[9] E. Gat. On three-layer architectures. In D. Kortenkamp, R. P. Bonnasso, and R. Murphy, editors, *Artificial Intelligence and Mobile Robotics*, pages 195–210. AAAI Press, 1998.

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

[11] P. Maes. Situated agents can have goals. *Journal for Robotics and Autonomous Systems*, 6(3):49–70, June 1990.

[12] M. J. Matarić. Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation*, 8(3):304–312, June 1992.

[13] M. J. Matarić. Behavior-based control: Examples from navigaton, learning, and group behavior. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2–3):323–336, 1997.

[14] M. N. Nicolescu and M. J. Matarić. Experience-based representation construction: learning from human and robot teachers. In *Proc., IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pages 740–745, Maui, Hawaii, USA, Oct 2001.

[15] M. N. Nicolescu and M. J. Matarić. Learning and interacting in human-robot domains. In C. C. White and K. Dautenhahn, editors, *IEEE Transactions on Systems, Man, and Cybernetics,Part A: Systems and Humans, Special Issue on Socially Intelligent Agents - The Human in the Loop*, volume 31, pages 419–430. IEEE, September 2001.

[16] P. Pirjanian. 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, 1999.

[17] S. Russell and P. Norvig. *AI: A Modern Approach*. Prentice Hall, NJ, 1995.

[18] M. Steenstrup, M. A. Arbib, and E. G. Manes. Port automata and the algebra of concurrent processes. *Journal of Computer and System Sciences*, 27(1):29–50, Aug. 1983.

[19] B. B. Werger. Ayllu: Distributed port-arbitrated behavior-based control. In *Proc., The 5th Intl. Symp. on Distributed Autonomous Robotic Systems*, pages 25–34, Knoxville, TN, 2000. Springer.