THESIS REPORT

Master's Degree

The Use of Behavior Hierarchies for Controlling a Vision-Based Space Teleoperation Robot

by O. Seeliger
Advisor: J.A. Hendler

M.S. 96 -1

Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry

The Use of Behavior Hierarchies for Controlling a Vision-Based Space Teleoperation Robot *

Oliver Seeliger
Master's Thesis
October 13, 1995
Department of Computer Science
University of Maryland
College Park, MD 20742

January 5, 1996

Abstract

In this thesis I describe the use of *behavior hierarchies* based on "merging" two models of multi-layer architecture — the supervenience model of [18] and the subsumption model of [6]. The behavior hierarchy approach allows us to use the robustness of reactivity in behavior design. It also encourages the design of modular behaviors that can be reused or more importantly recalibrated in different situations. I argue that behavior hierarchies extend our ability to design and program effective solutions that combine reactive and goal-driven components, but do not require any explicit planning. This work is used for an implemented system in which the underwater robot SCAMP developed at the Space Systems Laboratory at the University of Maryland performs vision-based behaviors to relieve a human operator of certain tasks.

*This research was supported in part by grants from NSF(IRI-9306580), ONR (N00014-J-91-1451), AFOSR (F49620-93-1-0065), the ARPA/Rome Laboratory Planning Initiative (F30602-93-C-0039), the ARPA I3 Initiative (N00014-94-10907) and ARPA contract DAST-95-C0037.

1 Introduction

In this thesis, I present an object tracking tool for space telerobotics. The objective of this application is to allow a human operator of an underwater robot to track stationary or moving objects. While performing such a task manually requires the operator's full attention, the object tracking tool lets the system perform the tracking job semiautomatically thereby reducing the load on the human system user. The motivation for this project is part of an ongoing effort to reduce the costs of human operators needed in space telerobotics.

The approach taken to design and implement the object tracking system is based on behavior-based control of autonomous robots. More specifically the work is influenced by the subsumption architecture for robotic behaviors [6] and by the supervenience architecture [19] borrowed from reactive planning. I argue that the use of behavior-based control combined with elements from supervenience, in which components are arranged depending on their epistemological closeness to the real world, facilitates the construction of systems such as for object tracking. Low-level components perform simple tasks and interact with higher-level components that configure the low levels and interface with the human operator. This arrangement permits the design and implementation of fully autonomous and semiautonomous systems that interact with the user.

In addition, the tracking of objects requires the use of vision. However, the construction of a complete world model according to traditional methods in computer vision is too expensive computationally to be of any use in a real-time robotic system whose robustness depends on a fast execution loop. As a result, the work presented in this thesis uses the approach of *purposive vision* [3]. Instead of constructing complete world models and extracting task-relevant features from that model, purposive vision uses information about the task and constraints on the environment to produce a more *task-specific* approach. The result is a significant speed-up needed for the real-time tracking system.

Figure 1 shows the underwater robot SCAMP seen in the lower right corner tracking the porthole at the top of the image. The upper left corner displays the robot's view of the same scene. There the target object, in this case the porthole, appears as a round region in the image center, which is marked with a square. While active, the tracking system attempts to keep the target object in the field of view by generating the appropriate motion control commands.

Section 2 describes the object tracking task and how it relates to ongoing efforts in space telerobotics to reduce the role of human operators to increase productivity in space missions. Section 3 then introduces the concept of *Behavior Hierarchies*, the architecture used to design and implement the tracking application. Section 4 presents the central motivation and key ideas behind purposive vision used to expoit task- and environment-specific features to simplify the perceptual process and thereby reducing its computational costs.

Then, I describe the object tracking application in detail. Section 5 contains the design of the system and relates it to the described research. In Section 6

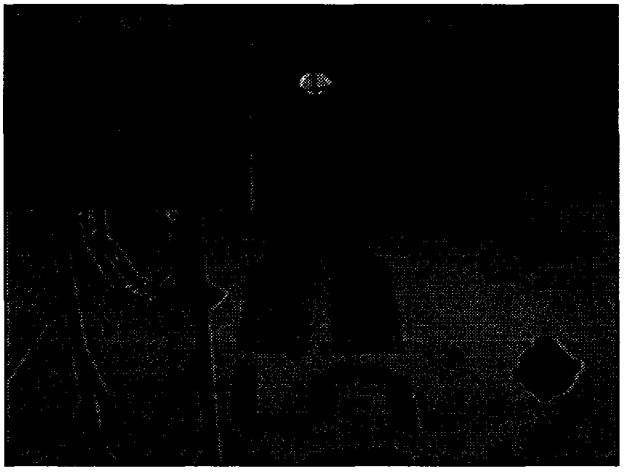

Figure 1: The SCAMP underwater robot tracking a porthole in the water tank of the SSL.

I elaborate on the actual implementation and in section 7 I summarize testing efforts, results, and conclusions.

Figure 2: SCAMP Underwater Robot

2 Object Tracking Task

The objective of the object tracking system is to provide astronauts with a *smart camera* or more precisely with a robot containing a camera that can perform certain autonomous tasks such as tracking objects. Because the goal is to relieve the operator of tracking a target object in order to let him/her focus the attention on the primary task, for example, on repairing the object, the camera robot should be a secondary device. As a result, the interaction between the tracking system and the operator should be very simple such that the user can perform the primary task without being excessively interrupted by the secondary tracking system. The degree of the system's autonomy should be high enough to allow the operator to concentrate on something else, and at the same time the tracking system has to be configurable in order to satisfy the operator's needs.

Figure 2 shows a picture of the robot for which we are building the object tracking system. SCAMP, the *Supplemental Camera and Maneuvering Platform*, has been built by the Space Systems Laboratory at the University of Maryland to provide robot operators with an independently moving camera view for supporting tasks in space. To simulate the weightlessness of space, SCAMP functions underwater. SCAMP is operated from a control station and

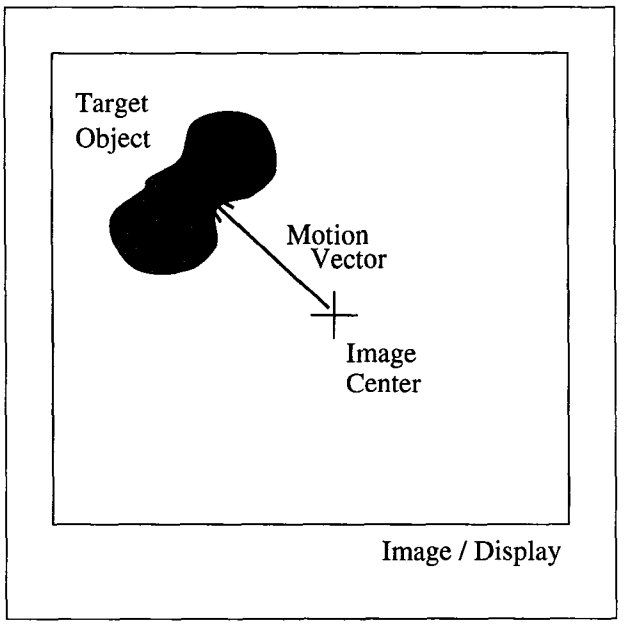

Figure 3: A simple representation of the tracking task.

communicates with it via a fiber optic link. The on-board microcontroller is a Motorola 68HC11, and the effectors consist of six fan thrusters. Details of SCAMP and its usage can be found in [9].

The tracking application is intended to run on the control station and serves as a communication link to the robot by interpreting incoming images and generating motion control commands. In addition, the tracking tool provides a user interface facilitating semiautomatic operation. The interface of the tracking system highlights an object previously selected by the operator as well as other objects with similar visual features.[1] In response to a selection, the system attempts to center the target with respect to the display by generating appropriate motion vectors as shown in figure 3. To configure the system, the user can either select any of the objects presented by the system or select another point or region, which will define new features for the system to extract. On important events, such as the disappearance of tracked features from the image or the appearance of new objects with the same features competing with the object currently tracked, the tracking system notifies the user by means of an alert.

The structure of the object tracking task is inherently hierarchical. More

[1] An object's features are defined in terms of its colors. See section 5 for more details.

specifically, the hierarchy arranges components based on their closeness to the real world, i.e. on how concrete the sensory elements are. The lowest components operate in terms of thousands of pixels and colors, which are very concrete elements, whereas the highest components depend on the more abstract notion of *objects* in order to reduce the amount of information the operator has to interpret. This is very crucial because of the tracking tool's semiautomatic flavor. As a result, the architecture of the object tracking system should be hierarchical in nature thereby achieving a smooth link between abstract, user-oriented components at higher levels and concrete, sensor-driven components at lower levels. This hierarchical structure makes this task particularly suitable to approaches based on behavior hierarchies, an architecture for robotic behaviors.

3 Behavior Hierarchies

Behavior-based robot control works by "hard-wiring" solutions to problems rather than by deliberating about a choice of actions or searching a state space. This feature allows a behavior-controlled robot to respond to its environment quickly by adequately reacting to changes. Typically, this reactivity is achieved by tightly coupling sensors to actuators through simple computations. More complex behaviors are constructed by structuring the system based on tasks rather than functions in the traditional *perception, planning, action* methodology.

3.1 Previous Work

Brooks' *subsumption architecture* [6] incorporates these features. In subsumption, every behavior couples sensors to actuators avoiding abstract representations, and progressively more complex behaviors are patched on to previously existing behaviors such that the resulting system *subsumes* the previous system. This results in a hierarchy of behaviors in which higher-level behaviors *arbitrate* lower-level behaviors by *inhibiting* their outputs and *suppressing* their inputs. Similar behavior-controlled systems have been implemented by [4, 11, 14].

While the reactivity of behavior-based systems is suitable for robots that have to survive in a rapidly changing and uncertain environment, there still is a demand for architectures that allow a greater extent of problem solving abilities. The need for both reactive as well as deliberative behavior has created the term *Reactive Planning*. A lot of work has been devoted to this field [1, 13, 16].

One approach to reactive planning has been the *supervenience architecture* [18, 19]. In this multilevel architecture, components at lower levels are epistemologically "closer to the world" than higher levels. While lower levels sense the world and report up sensed information in increasingly higher degrees of abstraction, higher levels *depend* on lower levels for world-knowledge. This dependency relationship stresses the importance of lower levels and their authority based on being more "in touch" with the real world. Higher levels on the other hand are more informed about the system's goal or task. Based on the information reported up, higher levels *configure* lower levels thereby adapting them to the system's goals. Configuring can include functions such as activating, deactivating, changing and monitoring parameters and modes as well as dynamically adding and removing behaviors to reformulate goals at higher levels to more specific actions to be carried out by child behaivors. This interaction is summed up in the phrase "world-knowledge up / goal-knowledge down" which guides planning and reacting in the supervenience model.

Based on the subsumption and supervenience architectures, I developed the concept of behavior hierarchies. The subsumption architecture, where high-level behaviors are only allowed to gate, i.e. turn on and off low-level behaviors, is more restrictive than allowed in using supervenience. On the other hand, the

development of the supervenience architecture was performed using a simulator which made many simplifying, and unrealistic, assumptions about the robotic behaviors which were available. Behavior Hierarchies are designed to overcome the limitations of the subsumption architecture and to apply the theoretical concept of supervenience to robotics.

3.2 Basic Features of Behavior Hierarchies

The central design feature in behavior hierarchies is borrowed from that of supervenience – the separation of behavior components based on "world-knowledge up / goal-knowledge down". Thus, the design of communication is such that the lower a behavior is in the hierarchy, the closer it should be to the world.[2] Naturally, the closer a behavior is to the world, the more concrete are the elements the behavior senses or manipulates. By contrast, the higher a behavior is in the hierarchy, the further away it is from the real-world and the more abstract are the behavior elements. In turn, "goal-knowledge down" requires that behaviors at a higher level are, essentially, closer to the goals of the system rather than anchored in particular sensory values. As goal-based information is passed to lower behaviors, it is decomposed and reformulated to result in a description closer to the world than the previous one. In this process goal-knowledge loses importance while world-information becomes more important.

Behavior hierarchies are extensions of subsumption architectures which assume a collection of behaviors not sharing any global variables. Instead, interaction between behaviors occurs through "wires" (typically implemented in software, rather than hardware). Each behavior has input and output wires, and output wires of behaviors are connected to input wires of other behaviors. All behaviors run concurrently sending back and forth information as necessary. Behaviors may activate and deactivate other behaviors according to the system's goals, and unlike the subsumption architecture they can also configure lower levels by changing their modes and parameters.

Behavior hierarchies work by imposing a partial order on the collection of behaviors. In this order, every behavior that precedes another behavior is, essentially, more abstract than that other behavior, i.e. more distant from the world (and, in fact, it cannot directly sense the world, but only receive inputs from the behaviors closer to the world). The communication between the components allows higher behaviors (those preceding others in the order) to send messages which either activate, deactivate, or change modes of behaviors lower in the hierarchy as shown in figure 4. Lower behaviors send messages which report "events" to higher behavior components. These events typically do *not* consist of specific sensor values, but rather values representing initiation, completion, or errors arising within the behavior. For example, a particular event

[2]Note that while this is typical of subsumption architectures, it is not mandated in the design, and is often violated in complex subsumption-based systems.

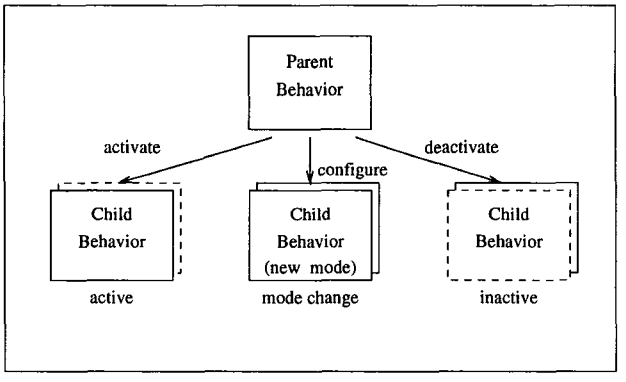

Figure 4: Basic Top-Down Communication Methods in Behavior Hierarchies

might represent that an obstacle was encountered while moving in a certain direction.

We should note here that unlike many "3-layered" behavior-based systems such as [11, 14], behavior hierarchies intentionally blur the notion of layer or level. No two behaviors are explicitly "on the same level." The only defining characteristic of a behavior hierarchy are the relationships explicitly defined by the partial order. The communication protocols, which allow a behavior higher in the hierarchy to request modifications of goal-oriented parameters of another lower behavior, but which only allow the lower to "inform" the higher, realize the asymmetric leveling which characterizes supervenience.

In the long term, behavior hierarchies are intended to be the lower part that is epistemologically closer to the world in the global supervenience architecture including more complex planning behaviors than those in the current implementation. Thus all behaviors in the current behavior hierarchy are relatively close to the world. This means that typically the level of the abstractions and therefore the need for symbolic representations of the world is limited. Symbolic abstractions are not supported nor encouraged in Brooks' subsumption architecture. To allow the behavior hierarchies to take advantage of the strengths of reactive behaviors, we also try to refrain from using explicit symbols wherever possible. Rather, the robot's environment is defined indexically, i.e. in terms of its relationship to the robot such as "the object that the robot is tracking".

It should also be noted that an explicit aspect of the supervenience approach is that a problem should be resolved at the lowest level possible. Information about the event may then propagate up and eventually change goals (and therefore cause information to propagate down and change behavior), but that doesn't imply that the system stops while this occurs. Behavior hierarchies also allow this, letting lower behaviors continue to run until a new signal (to change

behavior or deactivate) is received from above. This feature is crucial for robots embedded in dynamic environments because lower behaviors can respond to discrepancies between expected and actual events reactively without incurring a delay due to having to "consult" behaviors higher in the hierarchy.

The authority given to behaviors at lower levels in the hierarchy to react directly to unexpected events distinguishes behavior hierarchies from other abstraction-based robotics systems [2, 11, 17]. The key difference is that in response to unexpected events, lower behaviors can choose to take a course of action that differs from instructions by parent behaviors, but is more appropriate for the particular situation. In the extreme case, lower behaviors may not execute the instructions given by a behavior higher in the hierarchy or even do the opposite to ensure the robot's safety.

Another feature extending behavior-based systems is the *dynamic* execution structure of behavior hierarchies. Behavior hierarchies can dynamically add and remove behavior components depending on the robot's task and environment. I discuss this characteristic in more detail in the following section.

3.3 Dynamic Behavior Hierarchies

Many domains, in particular those requiring the use of vision, involve a very rich, potentially unbounded set of sensory events such as the information contained in a sequence of images. As a result, systems solving complex tasks need to continuously select information that is relevant to the current situation. Dynamic behavior hierarchies satisfy the need for information selection by allowing parent behaviors to add new and remove old child behaviors on a dynamic, as-needed basis in order to adapt to the changing information relevant to the task.

Dynamic behavior hierarchies achieve this by using behaviors in a manner similar to *monitors* as defined in [16, 10]. Whenever a new piece of relevant information appears, a behavior can add a new child behavior, configure it for monitoring the newly arrived sensory impulse, and activate the new behavior. Similarly a parent behavior can remove child behaviors trying to monitor something that has disappeared or is no longer relevant to the task or situation.

Adding a behavior to the dynamic hierarchy requires spawning a new process dedicated to executing that behavior, and creating communication channels to previously existing parent and child behaviors. This is different from activating a behavior, which involves waking an idle process. By analogy removing a behavior entails ending its process in contrast to deactivation causing the process to become idle.

Dynamic behavior hierarchies provide a more adaptive execution structure and more powerful ways to realize the supervenience paradigm. Since higher behaviors are more goal-driven and have to break down tasks into subtasks, adding and removing child behaviors dynamically constitute additional ways to perform this function. Behaviors lower in the hierarchies, on the other hand,

tend to sense world-knowledge more selectively, and adapting to relevant information dynamically therefore improves the quality of the world-knowledge propagated up. Hence the dynamic aspect of behavior hierarchies results in a powerful system for tasks requiring complex sensing.

How can one come up with new, useful behaviors to add to the hierarchy dynamically? This requires behaviors that are somewhat *general*, in the sense that some key parameters relevant to the monitoring task are left open at compile-time. For example, a behavior tracking objects of interest in an image sequence might have no initial values for the (x,y)-coordinates and the size of a tracked object. At runtime, when higher behaviors add copies of these monitoring behaviors, they configure the monitors by assigning initial values to the parameters to be monitored. In the example, every individual copy of the tracking behavior is configured before activation by specifying values for the coordinates and size of the object to be tracked. As a result, newly added behaviors can be thought of as instances of more generic behaviors.

However, additional behaviors involve a higher computational load, slowing down the system. To limit the number of behavior copies added to the behavior hierarchies, behaviors have to select more important copies from less important ones. The criterion for removing behavior copies is based on world-knowledge reported up by each copy. Based on goal-knowledge, behaviors higher in the hierarchy can evaluate the relevance of world-knowledge, and remove lower behaviors monitoring less important sensory information. The result is that dynamic behavior hierarchies control the load on the system by removing less important behaviors to compensate for the cost of newly added behavior monitors. In the following sections, I will show how the expressive and adaptive power of dynamic behavior hierarchies support all the requirements for the object tracking system.

4 Purposive Vision

Tracking objects visually requires processing image sequences captured by the robot's onboard camera. Many real-time vision applications such as the object tracking system depend on a fast execution loop for robustness and acceptable performance in critical situations. Robots operating in the "real world" have to be able to respond appropriately to situations in a dynamic world full of uncertainty. Therefore sensing the world (i.e. examining the images) often is essential and the key to avoiding fatal situations.

Traditional methods in computer vision advocate the construction of a general-purpose model of the world containing information such as egomotion-vectors and 3D-representations of objects encountered. This methodology treats applications of vision as "subproblems" of the general problem of visual reconstruction, and once the general problem has been solved, the application consists merely of extracting the application-specific information from the general representation. However, the designers of real-time systems are faced with the huge computational costs involved in this approach. Constructing a general world model is so time-consuming that it prohibits the fast execution needed to sense changes early enough for the system to react on time.

An active area of research directed at overcoming excessive computational costs is *purposive vision* [3, 12, 15]. The key is to extract only the information needed for the particular task rather than constructing a general-purpose model. In the object tracking task for example, we are mainly interested in the location of the target object such that the system can track it by issuing the appropriate motion control commands to position and orient the robot in the desired way. Since keeping the target object in the image center doesn't require determining its 3D position and orientation, the process of visual interpretation can be further simplified to image coordinates. The location and size of the object of interest in the image plane contains the information needed to issue control commands that will move the image center toward the target object and maintain a stable distance from it.

The centroid of the region of interest in the image plane indicates roughly the horizonal and vertical positions of the object relative to the robot. The size of the region of interest can be used to infer the distance between the object and the robot. Together these two pieces of information can be used to issue motion control commands to bring the centroid of a region of interest to the image center and to maintain a desired and stable distance between the robot and the object of interest.

Such task- and environment-tailored simplifications are not free, however. One penalty to be paid is reduced precision. The projection of the target object onto the image plane, for example, reduces the environment to the two dimensions of the image plane, and without reconstructing a three-dimensional model, it is impossible to perfectly maintain the same position and orientation relative to the target object. However many applications, including the object track-

ing system, don't depend on perfect precision; moreover, the actuators involved would not be able to match the precision of a perfect model recovery.

The other backdraw is of course less generality. The means by which simplifications are to be formalized require assumptions much like those used by Horswill in his Polly System [12]. As long as the assumptions hold, the simplifications of the visual recovery process will work well. However in the real world, it is hard to guarantee the correctness of the assumptions. For example, the object tracking application relies on distinct color features, and therefore one assumption is that the there is a sufficiently uniform surface on the target object that is also different in color so as to distinguish it from its background.

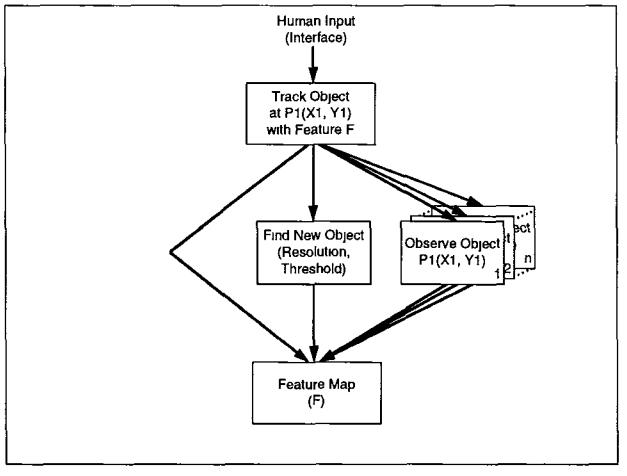

Figure 5: The Object Tracking Behavior Hierarchy

5 Design

In a supervenient hierarchy of behaviors, elements at lower levels are epistemologically closer to the world, i.e. more concrete, than elements at higher levels. This arrangement is similar to the process of visual reconstruction in the sense that the "input" consists of concrete images while the "output" is more abstract information derived from the input. As a result, lower levels in the behavior hierarchies operate on the images while higher levels reason about the input at increasingly higher levels of abstraction. Since the abstraction throughout the hierarchy is task-specific, the behavior hierarchy approach is particularly applicable to tasks requiring purposive vision.

5.1 The Behavior Hierarchy for Object Tracking

The behavior hierarchy for the object tracking task is represented as the partial order graph shown in figure 5. The arrows indicate which behaviors control other behaviors in the hierarchy. For example, the "Track Object" behavior controls "Feature Map", "Find New Object", as well as all instances of "Observe Object", and each of the last two behavior types also controls "Feature Map" at the bottom of the hierarchy. It should be noted that in this behavior hierarchy the human operator acts like a parent behavior to "Track Object" such that this top-level behavior functions as a user-interface by providing a means to be configured by the user at a sufficiently high level of abstraction.

The object tracking behavior hierarchy strictly follows the central paradigm

of forcing lower behaviors to be epistemologically closer to the world than higher behaviors, with higher behaviors dependent on them for sensing. This is illustrated by the fact that "Feature Map" senses and interprets *colors*, "Find New Objects" and "Observe Object" operate in terms of *features*, and "Track Object" reasons based on *objects*. While color can be sensed directly and is therefore close to the world, a feature is a binary abstraction dividing colors into positive "matches" and negative "rejects". Finally the notion of objects is based on clusterings of features and represents a further abstraction.

Since "Track Object" acts as a user interface, the human operator can completely configure all situation specific parameters such as the object to be tracked or modes such as approaching the object during tracking. Whenever the user requests a change of the system's mode, "Track Object" propagates that change to the lower behaviors in order to configure their behavior to the new needs. One such change occurs when the user picks a new object to be tracked by pointing at it on the display. The top-level behavior immediately configures "Feature Map" to extract the type of feature at the location picked by the user. At the same time, "Track Object" adds a behavior instance of "Observe Object" and configures it to inspect the area in the image where the user has clicked in order to track the object's features.

"Track Object" also activates "Find New Objects" whenever the system is in tracking mode. The purpose of this behavior is to look at the entire image at a coarser level to detect other regions with the same features as the tracked object. This information is not only useful for the operator to see what other regions compete for the system's attention and pose a threat by potentially distracting the focus from the object of interest, but "Track Object" can use it to relocate the object in case "Observe Object" loses track of it.

In addition to the top-level behavior, "Find New Object" and "Observe Object" also control "Feature Map" although their objective is primarily to monitor the sensory information generated by it rather than to configure it. This monitoring activity consists of searching for regions of positive features in the image in order to locate objects.

The instances of "Observe Object" illustrate the use of *dynamic* behavior hierarchies. The execution structure of the behavior hierarchy changes dynamically because the number of "Observe Object" behaviors varies depending on the number of regions with the color of interest. While one of the copies always monitors the object picked by the user so that "Track Object" can generate the appropriate control instructions to keep the object in the image center, the remaining copies of "Observe Object" monitor other objects with the same color. When the number of objects varies, "Track Object" activates and deactivates behaviors as needed. If too many behaviors are active for the system to support, "Track Object" removes behaviors to reduce the computational load involved in monitoring objects. In that situation, it picks the behavior tracking the lowest

feature intensity[3] as the candidate for removal.

In this manner, these four behavior types cooperate in achieving the object tracking task. Lets now look closer at each individual behavior part of the hierarchy.

5.2 The Individual Behaviors in the Hierarchy

As pointed out earlier, the lowest-level behavior "Feature Map" operates on the raw images from the camera input, while the behaviors "Define Feature", "Find New Objects", and "Observe Object" deal with a slight abstraction of an image, the *feature map*. The top-level behavior "Track Object" reasons in terms of target *objects*, a further abstraction that is based on feature maps. To mirror the process of vision, I will explain the behaviors in the order of increasing abstraction.

A behavior in behavior hierarchies can be represented as an object embodying parameters, actions, and methods for configuring and monitoring them. Only behaviors higher in the hierarchy have access to the methods of lower levels allowing them to manipulate lower behaviors as needed. Since every behavior adds its child-level behaviors to the hierarchy as needed on a dynamic basis, every behavior has an "add" method that initializes the behavior itself in addition to an "execute" method that defines the behavior's actions executed repeatedly in a loop. Conceptually lower-level behaviors can be thought of as parameters of higher behaviors, but more intuitively parameters represent values such as thresholds and tolerances. Other behavior elements are methods to configure, monitor, activate, deactivate, and remove the behavior.

5.2.1 Feature Map

The objective of the "Feature Map" behavior is to convert an image at any given time to a *feature map*. A feature map is a two-dimensional array of binary values that indicate whether the feature of interest exists at any point (x,y) in the image. Therefore, this behavior has to break down the original color image to a raster with the same dimensions, but with only two values 1 and 0. In the sense that the notion of color is replaced with the concept of a *feature*, the resulting feature map passed to higher-level behaviors is an abstraction of the original image.

Every type of behavior has a set of parameters that are typically initialized to default values and configured by the behaviors themselves based on the world-knowledge sensed or configuration requests from other behaviors at higher positions in the hierarchy. The "Feature Map" behavior has the parameters *feature-color*, *threshold*, *difference-function*, *color-image*, and *feature-map*.

[3]The *intensity* consists of the size of the object in the image with the color of interest as well as the average color difference between the region and the color of interest.

As mentioned before, "Feature Map" uses *feature-color* as a feature discriminator to achieve its task. Every pixel in *color-image* whose color falls within a certain distance, *threshold*, to the color of interest is a candidate for a "positive" feature, and the corresponding pixel in the feature map *feature-map* receives the value 1. On the other hand, all other pixel values in the feature map are set to 0. This behavior manifests itself in the "execute" method of "Feature Map"[4].

The *feature-color* parameter designates the color of the object of interest. Higher-level behaviors set this parameter indirectly based on the image position selected by the human operator. Since the "Feature Map" behavior determines the color of interest by picking the color of the pixel at the location selected, the lowest behavior in the hierarchy completely replaces the notion of color with the concept of a *feature* thereby allowing higher behaviors to reason on a more abstract and simplified level.

To determine whether a color satisfies the criteria of the target object, the "Feature Map" behavior applies the *difference-function* to *feature-color* and the color at every point in the image. If and only if the result of this operation is less than the *threshold* parameter, a positive feature is marked in the feature map. Both parameters are initialized to default values and unlike *feature-color* never change.

It is possible to modify *threshold* and *feature-color* on a dynamic basis, thereby making the system more adaptable to changes in the environment. Adapting *threshold* means changing the selectivity of the color discrimination process, and modifications of *feature-color* would allow the system to compensate for slight color changes of the target object due to differences in lighting conditions over time. However, since the system is intended to be a secondary device for a human operator, it is preferable to consult the operator on such events rather than risking undesired changes of the system's behavior.

When "Feature Map" is added to the hierarchy by a parent-level behavior, the "add" method is executed. The two parameters *color-image* and *feature-map* contain the storage for the input color image at every point in time and the output feature map.

Two other very important method types mentioned before are configuration and monitoring methods. By means of the "set-feature-color" configuration method, the behavior is told to monitor the color at a given point (x,y) in the image. The invocation of this method effectively changes the behavior of "Feature Map" by configuring the *feature-color* parameter.

Monitoring methods such as "get-feature-map", on the other hand, provide a way for other behaviors to obtain access to the results generated by lower behaviors.

[4]See appendix A for the pseudo code.

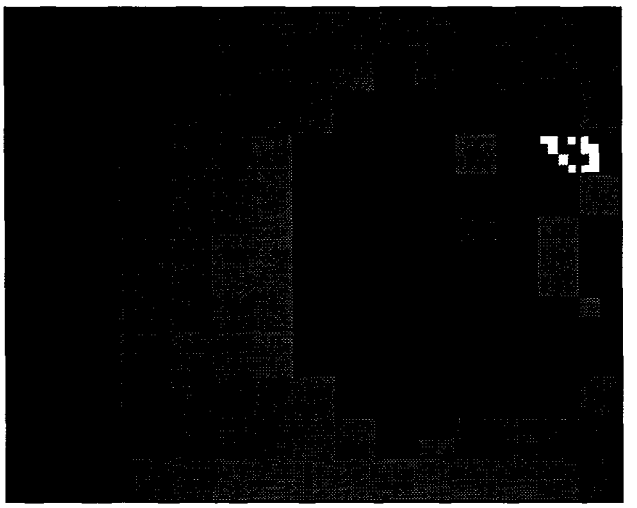

Figure 6: "Find New Objects" looking for feature matches at a coarser resolution.

5.2.2 Find New Objects

The objective of the "Find New Objects" behavior is to locate new objects in the feature map. To achieve this task, this behavior monitors "Feature Map" in order to find clusterings of positive features. Note that this terminology completely avoids the notion of *color* by replacing it with the more general concept of a *feature*. "Find New Objects" makes all *objects* found in the feature map available to its parent behavior. Since the image resolution affects the amount of information and thus the computation involved, "Find New Objects" interprets the feature map at a coarse resolution. The motivation is to save computation time by focusing the behavior's attention to the areas in the feature map containing positive features.

The "Find New Objects" behavior achieves this by superimposing a grid onto the feature map and examining the feature map selectively only at points lying on the grid. In case of a negative feature at a particular point, the next location is examined, while a feature match involves refining the grid at that location. As a result, the final representation depicts objects with the feature of interest with more detail than the "background". Figure 6 and figure 7 depict the result of this process. Note that the former represents the same scene as the latter, but at a coarser resolution by increasing the resolution only where signs of a feature match are located.

The process of inspecting the feature map can also be thought of as dividing

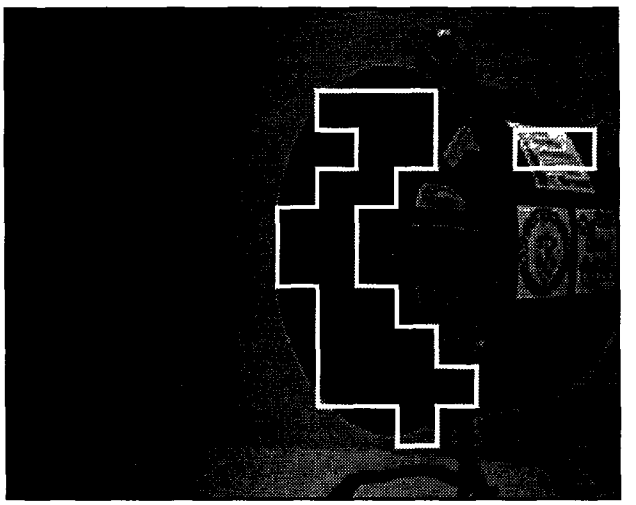

Figure 7: Display generated by "Find New Objects" for the user interface.

the matrix into squares whose sides are perfectly aligned. During the inspection, "Find New Objects" examines the feature map at the center of each square, and the corresponding feature reading is considered representative for the entire area. Consequently, a negative feature results in that square being rejected as a whole. On the other hand, a positive match is grounds for further refining the search for features by recursively subdividing the appropriate square until either a negative feature is hit or the resolution reaches a certain maximum.

Lets now look at the actual design of "Find New Objects". This behavior has the parameters *feature-map*, *resolution*, *min-resolution*, and *objects*. The *feature-map* parameter contains a link to the "Feature Map" behavior, which is needed to monitor the contents of the input feature map. In addition to the contents of the feature map, the parameters *resolution* and *min-resolution* define the behavior of "Find New Objects" by setting the resolution boundaries. The *objects* parameter denotes the output and consists of a set of regions with positive features, i.e. "objects". The "add" method initializes the last three parameters then used by the "execute" method.

In addition to these basic methods, "Find New Objects" has methods to configure the resolution parameters, to set the "Feature Map" behavior needed to obtain the latest feature map, and to monitor the set of objects found.

Parent-level behaviors invoke these configuration methods to set the instance of the "Feature Map" behavior that "Find New Objects" should use and to adjust the resolution parameter by requesting increases and decreases. The resolution configuration methods can be used when the number of objects are

outside a certain range. For example, when "Find New Objects" does not find any objects using the current resolution, a parent behavior can decide to increase the resolution in order to obtain a finer search of the feature map, which is more likely to discover smaller objects. On the other hand, the resolution can be decreased whenever the search process finds too many objects because in that situation focusing on the largest objects picked up by a coarser grid is more important. As a result, these three methods provide a means to configure the "Find New Objects" behavior from a more abstract level.

In addition, the behavior has a method for monitoring the *objects* parameter needed as input by the parent behaviors. That method simply returns the most recent set of objects found in the feature map such that the behavior calling the "get-objects" method can carry on the process of visual reconstruction on a higher level of abstraction. An "object" can be represented as an orthogonal polygon, a point denoting the center of mass in the feature map, or a bounding box.

5.2.3 Observe Object

One inherent weakness of the feature detection process used by the "Find New Objects" behavior is that the relatively coarse inspection of the feature map results in a fairly unsteady representation in which objects change locations and shapes often. Therefore, the output is only suitable as "starting points" for tracking, but not for the tracking process itself. Once an object has been identified in the feature map, a more stable tracking module is needed to monitor exclusively the region containing and neighboring the object of interest. The "Observe Object" behavior performs that function.

"Observe Object" needs to be able to deal with a variety of situations. First and most importantly, the behavior has to keep track of the object's location because both the robot camera as well as the object of interest are expected to move. The relevant parameter can be a point in the feature map indicating the center of mass of the object's projection. While this representation of a "location" contains no depth information, the two-dimensions in the feature map suffice for the tracking task. This is a clear example of what it means to extract only information needed rather than building a complete model first.

In addition, the "Observe Object" behavior needs to monitor the size of the object's projection in the feature map. Since the behavior only inspects the feature map within a certain region containing the object, the region has to be adapted to the size of the object. If the inspected region becomes too large, the portion of interest within the image is insignificant resulting in poor tracking performance. Also if a large part of the region is not covered by the object of interest, the tracking process is more prone to be distracted by other positive feature within the region not belonging to the object of interest.

On the other hand, if the region is too small, the "Observe Object" behavior only looks at part of the entire object. Consequently the estimated location

Figure 8: A rectangle can model and adapt to a variety of shapes.

in the feature map can be incorrect because the center of all positive features within the region is likely to differ from the center of mass of the entire object. For the tracking task this error condition would be particular critical since motion is involved, and therefore the region can "float" within the projection of the object, if it is larger than the region. As a result, "Observe Object" has to adjust the dimensions of the region it inspects in order to overcome these error-conditions.

The size of the region thus has to be continuously adjusted to fit the object of interest as closely as possible. One possibility is to use the minimum bounding rectangle, i.e. the rectangle with the smallest possible area that contains all object points in the feature map. However, this strategy can result in very large rectangles because it facilitates expanding the rectangle borders along an entire dimension even for a single pixel with a positive feature. For similar reasons, the maximum rectangle enclosed in the object is not suitable either.

The solution is to pick a rectangle and enlarge it dynamically whenever the border contains sufficiently many positive features. When the number of positive features falls below a certain number, the process shrinks the rectangle border. This process is repeated for all dimensions, and as such it is generalizable to an arbitrary amount of dimensions.[5] This process of iterative refinement is also more appropriate for a changing environment because it is decomposed into many small steps that each operate on the most recent state of the feature map. Figure 8 shows how the "Observe Object" works in practice by adjusting the rectangle's location, shape, and size to the current position of a diver.

The parameters of "Observe Object" are *x*, *y*, *size-x*, *size-y*, *feature-map*, *active*, *min-size*, *border-size*, *step-size*, *expand-threshold*, and *shrink-threshold*. The first four parameters define the object's perceptual dimensions in the feature map, and the *feature-map* is a link to the "Feature Map" behavior. The *active* parameter determines if the behavior is active or not while all remaining parameters describe the process of size adaptation.

The (x,y) parameters define the center of mass of all positive features within

[5] In practice, this approach has proven very useful and adaptable to change in the environment. Typically the object of interest is marked fairly accurately providing an intuitive way to mark objects of interest for the user interface.

the rectangle inspected. Thus, the behavior continuously recenters the rectangle at that point thereby updating the location of the tracked object. The parameters *size-x* and *size-y* hold half of the x and y lengths of the rectangle's sides such that (*x-size-x,y-size-y*) and (*x+size-x,y+size-y*) are the bottom-left and top-right corners of the rectangles.

To compute the center of mass, the "Observe Object" behavior maintains two one-dimensional arrays *i* and *j*, one for each dimension. Each positive feature lying on a sample point within the rectangle results in both arrays being incremented at the point's x and y positions. Consequently, the arrays not only show the range of the object of interest (i.e. a minimum bounding rectangle), but also the distribution needed to compute the center of mass.

Five parameters are devoted to the size adaptation process. The *min-size* parameter specifies the minimum size that *size-x* and *size-y* are never allowed to fall below. Even if the size of the target object falls below a certain size, it is not advisable to shrink the rectangle in order to allow the system to react to brisk motions that would move the object entirely out of a small rectangle. Then the behavior could lose track of the object.

The *border-size* parameter defines the size of a "frame" along the sides of the rectangle that is completely contained within the rectangle. Depending on the concentration of positive features inside this frame, the "Observe Object" behavior can decide to increase or decrease the rectangle. Whenever such a modification occurs, the behavior increments the appropriate dimension by *step-size* numbers of pixels in the feature map. The two parameters *expand-threshold* and *shrink-threshold* define two thresholds that indicate at which concentration of positive features the rectangle should be expanded and shrunk. Now we are ready to look at the algorithm for the "add"-method of "Observe Objects".

The "execute" method consists of two parts. In the first part, the (*x,y*) location is updated, and the second part consists of adapting the size of the rectangle, i.e. the parameters *size-x* and *size-y*.

For the "Observe Object" behavior, the configuration methods are particularly important because they determine key parameters such as the object to be tracked by the system as a whole. They are "set-point", "reset", and "set-feature-map-behavior". After the former is used by a parent behavior to mark an object in the feature map containing the point selected, an invocation of "reset" results in the size parameters to be initialized to the minimum size. The "set-feature-map-behavior" method usually is called only once for each instance of "Observe Object" and creates a link to the feature map to be used by this behavior as input.

Since parent-level behavior need to access the tracking information generated by the "Observe Object" behavior, the monitoring methods are also essential. One method to extract this information is "get-point" which simply returns the feature-map coordinates of the rectangle enclosing the tracked object. The "get-rectangle" method, on the other hand, returns the entire rectangle, and

parent behaviors only interested in the area can use the "get-area" method.[6] For the tracking task, however, the most important monitoring method is "get-direction" returning a two-dimensional vector that represents the direction in which the tracking system has to move, in order to maintain the object's center position in the image. With these methods, the "Observe Object" behavior performs the *visual pursuit* task that lays the ground work for object tracking to be performed by the parent behavior "Track Object".

5.2.4 Track Object

The top-level behavior "Track Object" not only coordinates the other low-level behaviors, but it also functions as a user interface. The coordination activity entails adding, configuring, and monitoring all three child behaviors. Before "Track Object" starts executing its behavior, it adds an instance of "Feature Map", "Find New Objects", and "Observe Object" to the behavior hierarchy, and configures the communication channels between the behavior. The latter task involves for example informing the "Observe Object" behavior, which instance of "Feature Map" to access as its child behavior.

Functioning as a user interface implies that configuration and monitoring methods are also needed for "Track Object" such that the human operator can interact with the system. Since this behavior reasons in terms of *objects* at a high level of abstraction familiar to the user, the interaction between the tracking system and the operator is very natural. The user can specify requests in terms of objects on the display, and the tracking system can present the state of the system by highlighting objects on the display. First I am going to address "Track Object"'s role as the top-level behavior in the behavior hierarchy, and then I will discuss the user interface.

The objective of "Track Object" is to track the object of interest by generating control commands to move the object to the image center. The current location of the target object is determined by the "Observe Object" behavior that makes the position in the image available to "Track Object". Whenever the operator requests a new target object to be tracked, "Track Object" reconfigures "Feature Map" to extract the new color of interest as well as "Observe Object" to monitor the new initial position.

In order to access its child behaviors as needed during the tracking process, "Track Object" adds an instance of "Feature Map", "Find New Objects", and "Observe Object" in its "add" method. Furthermore, this method configures the latter two behaviors such that they access "Feature Map" as their child behavior. This activity can be though of as establishing communication channels between the child behaviors to allow them to interact properly.

[6] The usefulness of this piece of information will become apparent when we look at the design of "Track Object", which uses the area of the rectangle marking the tracked object in order to obtain some notion of distance between the tracking system and the object.

In addition to the *feature-map*, *find-new-objects*, and *observe-object*, which all point to child behaviors, "Track Object" has the parameters *point-selected*, *depth-mode*, and *area*. The former indicates the last point in the image selected by the user; this parameter is needed by the configuration method allowing the operator to select a new object of interest. The *depth-mode* parameter is another configuration parameter indicating whether the tracking system is stationary or moving toward or away from the object of interest during tracking. Furthermore, *area* defines a size estimate of the object in terms of its appearance in the image serving as measurable feedback to the "move-closer" and "move-away" modes.

The "execute" method consists of issuing control commands based on the position of the target object in the image. The implementational issues regarding control are explained in section 5.7. In addition to that, the "Track Object" behavior needs to check if the user has selected another object of interest in order to reconfigure its child behaviors.

The interface of the "Track Object" behavior consists of two parts. First, the behavior needs to respond to configuration requests from the operator. This is done by means of the methods that configure the *depth-mode* parameter (i.e. stationary tracking, move closer or move away) and select a new object. The second part of the interface consists of displaying the situation in a manner comprehensible to the user by marking the tracked object on the display. In this fashion, the "Track Object" functions as a user interface.

5.3 The Object Tracking System

This section describes the actual implementation of the object tracking system with the objective of providing a specific working example as well as to document the system for future work. The tracking application runs on a Macintosh Quadra 630 although it should be easily portable to other Apple Macintosh models with a video card. The computer is connected via a fiber-optic link to the underwater robot SCAMP built by the Space Systems Laboratory at the University of Maryland. This robot has six fan thrusters allowing it to move in 5 dimensions[7]. The data link sends the video output as well as control feedback from the on-board Motorola 68HC11 to the computer and control commands in the opposite direction. This constitutes the basic hardware of the object tracking system.

Lets now look in more detail at the software implementation of the system and more specifically at the implementation of each of the behaviors in the hierarchy.

5.4 A Behavior Hierarchy as C++ Objects

Applying the object-oriented methodology to behaviors has two main advantages, encapsulation and inheritance. Encapsulation insures that certain parameters that are privately owned by a behavior cannot be accessed directly by other behaviors, and therefore such parameters have to be accessed indirectly via method calls. While this might seem restrictive, this mechanism serves as an additional check that the supervenience paradigm holds. In particular, the behavior "objects" are to be arranged such that every behavior can only modify goal-related parameters and monitor the "world-knowledge", i.e. results, of other behaviors that are epistemologically closer to the world, i.e. lower in the behavior hierarchy.

Lets look at the example of the "Observe Object" behavior. This behavior can only monitor the "Feature Map" behavior because that is the only behavior lower in the hierarchy. In fact, the means by which "Observe Object" has access to "Feature Map" is a pointer to the corresponding C++ object. The parent-level behavior "Track Object" can not be accessed by "Observe Object" because of its higher, more abstract level in the hierarchy.

It should be noted that the pointer representation of behavior links emphasizes the *dependency* relationship in supervenience. In particular, higher-level behaviors require pointers to lower-level behaviors for sensing. While the lowest behaviors in the hierarchy are completely "self-contained", i.e. operate without accessing other behaviors, higher behaviors constantly monitor and modify lower behaviors. The pointers mirror this dependency.

Another more theoretical benefit of the object-oriented terminology is inheritance. Conceptually a behavior is represented as a "general" object class

[7] X, Y, Z, yaw, and pitch.

that has an "add" method initializing the behavior and allocating memory in addition to an "execute" method whole code implements actions. In addition to the behavior class we can think of certain behavior types. Since these behavior types are less general, but all satisfy the constraints for general behaviors, they can be thought of as behaviors that inherit certain qualities from the general behavior class. While inheritance can be applied at various levels of generality, this project only uses two levels, the general behavior class and a set of behavior type classes.

As a result, inheritance allows the behavior hierarchy designer to specify very expressive restrictions as to what behavior types a certain behavior can access. Suppose for example "Feature Map" was one of several low-level behaviors sensing features[8]. Then these behavior types could be grouped in a slightly more general behavior class "Feature Sensor", and by allowing other behaviors higher in the hierarchy to access feature sensors, such behaviors can decide which child behavior to consult in a particular situation. Therefore, inheritance could prove useful for more complex behavior hierarchies.

5.4.1 Parameters

All parameters are represented as private data elements of objects. While it is possible to have public parameters in behaviors at the bottom of a behavior hierarchy, because the parameters can be monitored globally by any behavior, this would violate the supervenience paradigm as soon as modifications were made or in cases where multiple behaviors existed in "leaf" positions. Therefore representing parameters as private data elements serves as an additional check that the supervenience paradigm holds everywhere in the implementation.

5.4.2 Methods

As mentioned before every behavior has two public methods called "add" and "execute". The former method typically initializes the behavior itself by assigning default values for the parameters and allocating memory for complex data types, and it adds any needed child behaviors to the hierarchy. Since the "execute" method embodies the "actions" (code), it constitutes the behavior's main body. The "execute" method is invoked on a regular basis such that every behavior appears to be executing continuously. This mechanism functions by means of a "Task Queue".

[8] This scenario is comparable to having more than one sensor to obtain the same information such as infrared and sonar sensors for measuring distance. Such a case could apply when each sensor is more reliable in particular situations, for example infrared sensors are more effective for sensing untextured surfaces while sonar sensors are unaffected by darkness.

5.5 Task Queue

The system's task queue consists of a circular doubly-linked list. Every node in the list contains a *task-id* as well as a pointer to a behavior[9]. As a behavior is added to the hierarchy, a corresponding entry is inserted into the task queue. By analogy, the system deletes a node in the task queue whenever the behavior addressed by the node is to be removed from the behavior hierarchy. This way, the task queue supports adding and removing behaviors dynamically during execution.

The mechanism for adding a behavior by means of the task queue is particularly noteworthy. Since parent behaviors add all child behaviors that they need and that are not already in the hierarchy, the behavior hierarchy is built from the top down. In the tracking behavior hierarchy, the system adds the top-level behavior "Track Object" upon start-up. Then "Track Object" adds "Feature Map" in order to monitor its results later-on. Similarly the top-level behavior adds the "Find New Objects" and "Observe Object" behaviors.

5.6 User Interface

The user interface is maintained by the top-level behavior "Track Object" and consists of the four windows shown in figure 9. The top-left window shows the original image from the camera, and the top-right window displays a version of the same image consisting of fewer colors thereby simplifying the input. The contents of this window essentially depict the feature map generated by the "Feature Map" behavior, but in addition to positive features that are displayed in their original color, the window shows negative features in other distorted colors. The bottom-left window shows the color of interest as well as the control output in form of a two-dimensional vector. Finally, the window intended for user interaction is the one in the bottom-right part of the screen. It contains the original image in addition to various "markers" high-lighting objects with the color of interest. This window is generated jointly by the "Find New Objects" and "Observe Object" behaviors that both add their results to this window.

While this constitutes the "output" part of the user-interface that conveys information from the system to the operator, the other half consists of reading the user's instructions. The bottom-right window functions both as a display and as an input device. To select a new object in the image to be tracked by the system, the operator can click within this interface window. Such an operation has two effects. First, "Track Object" configures "Feature Map" to extract a new color of interest depending on the location of the user's selection. Second, the main instance of "Observe Object" is recentered at the new location, and the object size is reset to the default size[10].

[9] This is a subtle feature emphasizing the role of inheritance because in C++ the nodes can point to any behavior regardless of the derived behavior type.

[10] In the design, all instances of "Observe Objects" that track objects other than the object

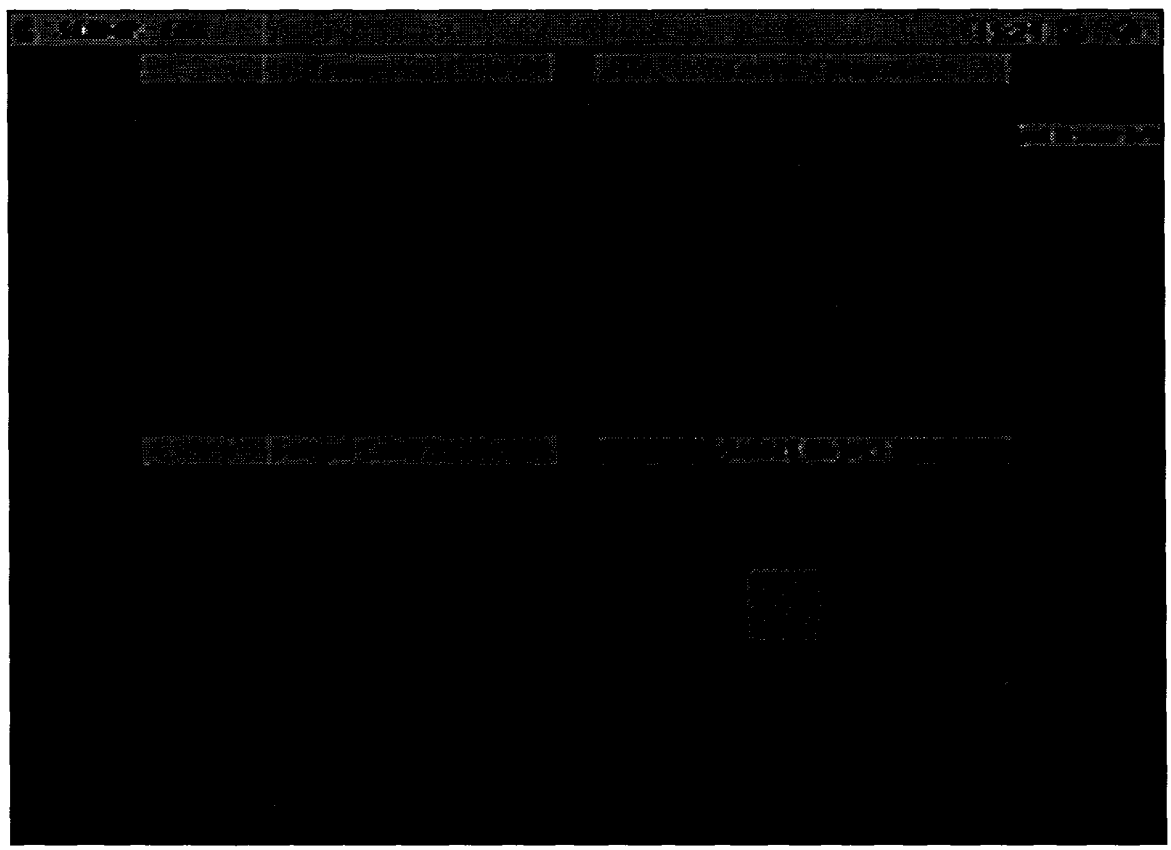

Figure 9: The user interface of the system during tracking.

The remaining parts of the user interface consist of a menu that allows the user to change the tracking mode of the system. This includes options to move the robot closer to or to move away from the tracked object. In addition to the user interface, "Track Object" also handles the motion control for the robot actuators.

5.7 Control

Based on the position of the target object in the image, "Track Object" generates motion control commands to center the object with respect to the robot camera's field of view. Since the behavior uses the image plane as a basis for tracking, this approach only generates motion vectors within two dimensions. These two dimensions can be mapped to translational and/or rotational motions such that the horizontal component of the object's position relative to the image center becomes effective as "Y-translation" or "yaw" and the vertical component translates to either "Z-translation" or "pitch". For the purpose of this project, "yaw" and "Z-translation" were chosen.

The very first attempt at building a control system consisted of a so-called "P-controller". Such a controller generates corrective motion proportional to the size of the displacement of the target object from the image center. There are two general problems with this approach. First, we observe "steady-state errors" that are due to the fact that the system doesn't generate enough velocity to correct such errors. The second problem involves oscillations due to the opposite reason; since the system generates high velocities, it tends to overcompensate and miss its target. This behavior results in oscillations around the target object. Although the system may keep track of the target object, such motions are highly undesirable.

The final system, a "PID-controller" addresses both problems. In this approach, the corrective motion consists of a weighed sum of three values. As in the "P-controller", one of them is a velocity proportional to the displacement (P). The second addend is proportional to the integral of the displacement within the last n displacement readings (I), and the third part consists of the displacement derivative (D). While the second component solves the steady-state errors by increasing the corrective motion, the third component eliminates oscillations by decreasing the motion vector.

The "Track Object" behavior uses the object location to derive a two-dimensional, corrective motion. In addition to that, the system can move closer to or away from the target object by using "X-translations". Since these motions can occur during tracking, the tracking system can be in three mutually-exclusive "depth modes": either stationary, "move closer", or "move away" mode. In "move closer" mode, the system approaches the object of interest

of interest are removed after each user selection in order to be replaced by new copies generated by the "Find New Objects" behavior. In the implementation, only one instance of "Observe Object" is allowed because of the high computational burden on our Quadra 630.

until it observes a 10% size increase of the object in the image plane. Similarly, the robot backs up in "move away" mode until the size of the object in the image plane shrinks by 10%.

Lets now look at the concrete representation of objects. As we have seen so far, such a representation needs to support the notion of object location and object size.

5.8 Representation of a visual "Object"

Since we characterize visual objects as they appear in the image plane, two-dimensional representations seem to be the most appropriate. One reason is that everything beyond the image plane is not grounded well enough in the real world according to the paradigm of [7] because a three-dimensional representation would have to include unreliable information such as the depth of the object[11]. This is also inconsistent with [18]. In supervenience, components at a higher level depend on lower levels for world-knowledge, and therefore this sensed information needs to be reliable. At the very least, the computation of depth should be explicitly marked as an abstraction by introducing a separate level thereby identifying the notion of depth as an unstable abstraction derived from lower-level world-knowledge[12]. For the purpose of keeping a target object in the field of view, two-dimensional representations of visual objects suffice.

One option is to model objects based on two-dimensional ortho-polygons which are rectilinear polygons made up only of right angles. This representation would be particularly appropriate for image rasters because of the arrangement of the image coordinate system in which pixels are organized into parallel rows and columns that are perpendicular to each other. As a result, such a representation could be applied at the pixel level such that every side of an ortho-polygon has a minimum length of one pixel or the system could apply the same concept at a higher level of granularity. In the latter, more general case, every side of an ortho-polygon would have a length of n pixels where n is a positive integer and a parameter that could be changed dynamically. Hence representing objects by means of ortho-polygons would provide accurate descriptions of regions in the raster image.

An inherent weakness of this representation is its complexity. Since ortho-polygons require reconstructing the shape of objects at every frame in addition

[11] Attempting to compute the depth of objects would result in unreliable information because in order to do that we need precise control of the robot to use an image sequence taken during a known rigid motion (known translation and rotation).

[12] Indeed, for applications requiring the computation of depth information, a new behavior "Depth Map" could be added to the "Object Tracking" behavior hierarchy. This new behavior could be added between "Track Object" and "Feature Map" or alternatively "Depth Map" could become a leaf-level behavior depending on whether the depth computation should be based on the original image or on features. The flexibility of arranging behaviors based on their closeness to the sensed world allows us to make such additions relatively easily without modifying the existing behavior construct.

to varying location and size parameters, the resulting representation is far too complex to be interpreted in real-time. Moreover, for tracking an object, the object's shape is less important than its location and size. One could argue that the operator would prefer a representation close in shape to the object, but when implemented the result was a flaky display due to the coarse modeling of objects[13]. In addition, the user-interface should be simple to allow the operator to grasp the state of the tracking system in a minimal amount of time. Therefore, the ortho-polygons are too accurate for our purposes.

A much simpler representation that also captures the notion of object location and size needed for tracking is the use of rectangles. The rectangle center indicates the location while the side lengths provide estimates for the object's size. Marking an object in the image results in a simple display that is easy for the user to understand and interpret in real-time. In addition, if the rectangle parameters are adjusted iteratively with each new frame, the display tends to be very stable thereby producing a much more continuous display. For these reasons, I represent visual objects as rectangles.

5.9 Visual Pursuit

The visual pursuit task performed by the "Observe Object" behavior involves maintaining an accurate representation of the the target object. In particular, this task involves updating the relevant parameters, i.e. the location and the size of the object in the image plane. Since these parameters are defined in terms of a rectangle, "Observe Object" manipulates that rectangle attempting to capture the entire object.

More specifically, "Observe Object" focuses its attention exclusively on the area within the rectangle. At each new frame, the behavior determines the center of mass of the object tracked within this rectangular image window, and based on that it repositions the rectangle such that its center coincides with the center of mass. This activity updates the system's model of the target object's location. In addition to that, the "Observe Object" behavior keeps track of the object size by monitoring the rectangle borders expanding and shrinking them to fit the object. With this method "Observe Object" visually tracks its target.

Since this approach is inherently iterative, the process not only converges quickly to a "correct" representation of the object based on the initial pointer provided by the operator, but it also functions well in dynamic environment in which objects tend to move rapidly or unpredictably. As a result, the system was not only tested on stationary objects, but also to moving targets. In particular, the system's ability to update its location quickly allowed it to keep track of moving objects.

[13]The coarse modeling of objects entails sampling image points very selectively. For ortho-polygons of granularity n, every pixel is considered representative for n^2 pixels, and as a result the sensed shapes vary significantly throughout an image sequence.

5.10 Color Discrimination

The "Feature Map" behavior converts color images to *feature maps.* To do this, it can apply different methods based on the red, green, and blue components of image points as well as a color of interest also represented as RGB components. The color discrimination process is general in the sense that the behavior uses a changeable color of interest in addition to a "color-difference" function. Such a function has two color parameters and typically returns low values for preferred colors, i.e. colors similar to the one of interest, and high values for color rejects. This color-difference function can express a variety of color discrimination methods.

For example, "Feature Map" could operate on black and white images by mapping both colors to intensity values and returning their difference. The method used for this project involved computing the sum of the RGB-component differences since this discrimination method is more powerful than simply using intensity differences. Another method explored was the use of *chromaticity.* Chromaticity can be approximated as the ratio of a particular RGB-component with respect to the sum of all components. This method attempts to eliminate differences in brightness from the color discrimination process[14].

While the result of applying a color-difference function to two colors is an integer typically in the non-negative range, "Feature Map" reduces this scalar to a boolean value that determines whether a positive feature has been identified at any given point. This conversion occurs by means of a *threshold* that determines the cut-off for the maximal color difference tolerable for a positive feature. This threshold can be modified dynamically to make the color selection process more selective or more tolerant depending on the amount of positive features identified.

5.11 Testing

The tracking system was tested in the water tank of the Space System's Laboratory at the University of Maryland over the course of four weeks. In particular, the testing effort involved calibrating the control system, improving the performance, and identifying boundary conditions to assess the system's robustness. One interesting result was that although the tracking system was originally designed to track stationary or slowly moving objects, we were also able to achieve reasonable performance tracking divers at relatively high degrees of difficulty. As can be seen in figure 10, the tracking system quickly adapts to changing situations.

The system can track objects at varying degrees of difficulty. The easiest scenario is depicted in figure 11 showing SCAMP (in the bottom-right corner) tracking a porthole. In this case the target object is sufficiently different in

[14] One advantage of characterizing colors by means of their chromaticity was the fact that this method corrected an imperfection in the robot's camera.

Figure 10: The diver's location and size are continuously updated.

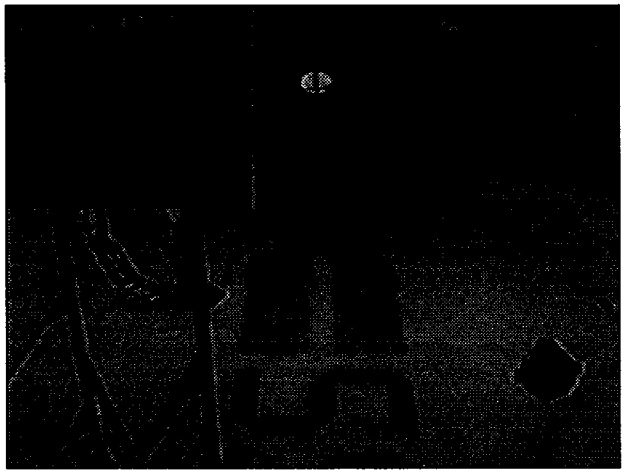

Figure 11: SCAMP (bottom-right corner) tracking a porthole.

color from the "background" that the system easily reifies the porthole. Slightly harder conditions exist when the target object is more similar to the background or when other objects with the same color appear in the image. In this case, there is the possibility for the system to get distracted in situations where two objects with the same color get very close to each other. One such example was when the system tried to separate a diver from his reflection occurring in the water surface.

The next level of difficulty involves obstructing the target object during tracking such that the feature of interest becomes occluded momentarily. In almost all the cases, the tracking system would relocate the target object after the occlusion. Perturbing the robot during tracking represents a yet more difficult scenario. In the worst case, physically pushing the robot out of position can result in the target object completely disappearing from the robot's field of view such that the object needs to be relocated. However, perturbing the robot

slightly, leaving the target object within the robot's sight usually allowed the system to correct for such outside interferences.

The hardest scenarios involve target objects that can move faster than the robot itself. In these situations the tracking system has to limit itself to keeping the target object, for example a diver, in the field of view rather than attempting to follow it. Since the robot yaws to compensate for horizontal displacements of the target object from the image center, the robot's maneuverability is impressive for "left" and "right" motions. Moving "up" and "down", however, is much slower such that the robot tends to lose targets that move at high vertical speeds.

In order to actually follow a diver, we had to use the "move-closer" mode of tracking. In this mode the system automatically orients the robot to face the target, and the operator can issue forward motion commands. Since such commands are at a relatively high level, i.e. "follow the target until it appears 10% bigger in the image", this "follow" mode is still semi-automatic in flavor. For example, when the target object suddenly stops, the robot will come to a halt, too, because the system automatically reverts to stationary tracking.

6 Conclusions

In this paper we have described the use of behavior hierarchies based on "merging" two models of multi-level architecture — the supervenience model of [18] and the subsumption model of [6]. As such, behavior hierarchies combine goal-oriented and reactive behavior. Higher behaviors drive the system to the successful execution of the task while lower behaviors provide the reactivity needed by the robot to survive in an uncertain and dynamic world.

We have shown that behavior hierarchies assign authority to higher levels like many other abstraction or hierarchy-based models [2, 11, 14, 17], but also to lower behaviors that are closer to the world. To support the former, goal-based authority, behavior hierarchies provide a rich set of means to decompose complicated tasks and to monitor their progress with respect to the system's goals. On the other hand, lower behaviors are more informed about the environment, and higher behaviors *depend* on them for sensing the world. Thus, behavior hierarchies encourage both careful sensing and abstracting.

In addition, it was demonstrated that behavior hierarchies facilitate behavior component reuse. Since behaviors can be configured by parent behaviors adapting them to the system's goals, the components are more useful in a wider variety of situations and tasks. Dynamic behavior hierarchies reuse behaviors by adding new copies at runtime and then configuring them to their specific needs.

To illustrate the usefulness of behavior hierarchies, we have described an implemented system. The object tracking system has served as an example for a complex task requiring vision. The system stresses information selection, as well as the adaptive features particular to *dynamic* behavior hierarchies. Future work could involve building and combining a larger repertoire of behaviors thereby exploring the scalability of behavior hierarchies. Another interesting direction is to connect behavior hierarchies to a symbol-based planner making high-level or search-oriented decisions.

A Pseudocode

A.1 Feature Map

```
behavior feature-map.add()
begin
   allocate(color-image);
   allocate(feature-map);
   feature-color := white;  // or any color for that matter
   threshold := 1;
   difference-function := exact-match;
end.

behavior feature-map.execute()
begin
   read color-image;
   for all (x,y) in color-image do
      if (difference-function( color-image(x,y), feature-color )
          < threshold)
         feature-map(x,y) := 1;
      else
         feature-map(x,y) := 0;
end.

behavior feature-map.set-feature-color(x, y)
begin
   feature-color := color-image(x,y)
end.

behavior feature-map.get-feature-map(x, y)
begin
   return feature-map;
end.
```

A.2 Find New Objects

```
behavior find-new-objects.add()
begin
   resolution := MAX-RESOLUTION;
   min-resolution := MIN-RESOLUTION;
   objects := {};
end.

behavior find-new-objects.execute()
begin
```

```
   objects := {};
   for y:=0 to (height - resolution) step resolution do
      for x:=0 to (width - resolution) step resolution do
      begin
         feature :=
          find-new-objects.examine-feature-map(x, y, resolution);
         if (feature = positive)
            objects := objects
             + (x, y, x+resolution, y+resolution);
      end;
end.

behavior find-new-objects.examine-feature-map(x, y, t-resolution)
begin
   if (t-resolution > min-resolution) and
      (feature-map[x + t-resolution / 2,
                   y + t-resolution / 2] = positive) then
      return (
         examine-feature-map(x, y, t-resolution / 2)
       + examine-feature-map(x + t-resolution / 2,
          y, t-resolution / 2)
       + examine-feature-map(x, y + t-resolution / 2,
          t-resolution / 2)
       + examine-feature-map(x + t-resolution / 2,
           y + t-resolution / 2, t-resolution / 2)
             );
   else
      return 0;
end.

behavior find-new-objects.increase-resolution()
begin
   if (min-resolution > 0)
   begin
      min-resolution := min-resolution - 1;
      resolution := resolution - 2;
   end;
end.

behavior find-new-objects.decrease-resolution()
begin
   if (resolution < MAX-RESOLUTION)
   begin
      min-resolution := min-resolution + 1;
```

```
      resolution := resolution + 2;
   end;
end.

behavior find-new-objects
   .set-feature-map-behavior(feature-map-behavior)
begin
   feature-map := feature-map-behavior;
end.
```

A.3 Observe Object

```
behavior observe-object.add()
begin
   allocate(i);
   allocate(j);
   min-size := 20;
   size-x := MIN-SIZE;
   size-y := MIN-SIZE;
   border-size := 3;
   expand-threshold := 0.8; // 80%
   shrink-threshold := 0.4; // 40%
   step-size := 5;
end.

behavior observe-object.execute()
begin
   // initialize local variables
   left := x - size-x;
   right := x + size-x;
   top := y - size-y;
   bottom := y + size-y;
   X := Y := M := feature-x := feature-y := 0;

   // compute center of mass
   for a:=0 to GRIDSIZE-1 do begin
      i[a] := 0;
      j[a] := 0;
   end;

   for a:=0 to GRIDSIZE-1 do begin
      for b:=0 to GRIDSIZE-1 do begin
         if (feature[left + a*(right-left)/GRIDSIZE,
                     top + b*(bottom-top)/GRIDSIZE])
```

```
         begin
            i[a] := i[a] + 1;
            j[b] := j[b] + 1;
            Y := Y + b;
         end;
      end;
      M := M + i[a];
      X := X + a*i[a];
   end;

   // 1. update location (x,y)
   if (M > 0) begin
      x := left + X/M * (right-left)/GRIDSIZE;
      y := top + Y/M * (bottom-top)/GRIDSIZE;
   end;

   // 2. update size (x-size,y-size)
   for a:=0 to border-size-1 do begin
      feature-x := feature-x + i[a];
      feature-y := feature-y + j[a];
   end;
   for a:=GRIDSIZE-1 to GRIDSIZE-border-size-1 step -1 do begin
      feature-x := feature-x + i[a];
      feature-y := feature-y + j[a];
   end;

   // adjust horizontally
   if (feature-x >= expand-threshold * border-size * GRIDSIZE)
      size-x := size-x + step-size;
   else
   if (feature-x <= shrink-threshold * border-size * GRIDSIZE)
      size-x := max(size-x - step-size, min-size);

   // adjust vertically
   if (feature-y >= expand-threshold * border-size * GRIDSIZE)
      size-y := size-y + step-size;
   else
   if (feature-y <= shrink-threshold * border-size * GRIDSIZE)
      size-y := max(size-y - step-size, min-size);
end.

behavior observe-object.set-point(new-x, new-y)
begin
   x := new-x;
```

```
   y := new-y;
end.

behavior observe-object.reset(feature-map-behavior)
begin
   size-x := min-size;
   size-y := min-size;
end.

behavior observe-object
 .set-feature-map-behavior(feature-map-behavior)
begin
   feature-map := feature-map-behavior;
end.
```

A.4 Track Object

```
behavior track-object.add()
begin
   // add child behaviors
   feature-map := feature-map.add();
   find-new-objects := find-new-objects.add();
   observe-object := observe-object.add();

   // create communication channels between child behaviors
   find-new-objects.set-feature-map(feature-map);
   observe-object.set-feature-map(feature-map);
end.

behavior track-object.execute()
begin
   if (content-selected(display))
   begin
      point-selected := get-location(display);
      reset-motors();

      // configure feature-map behavior for new color
      feature-map.set-feature(point-selected);

      // configure observe-object to track new object
      observe-object.set-point(point-selected);
      observe-object.reset();
      observe-object.activate();
```

```
      // reset depth-mode parameter for stationary tracking
      depth-mode := NO-DEPTH;
   end;
   else if (observe-object.is-active())
   begin
      point := observe-object.get-point();
      direction := observe-object.get-direction();

      // generate motion vector (horizontal,vertical,depth)
      if (depth-mode == NO-DEPTH)
         move(u, v, 0);
      else if (depth-mode == MOVE-CLOSER)
         // move forward until the area grows to 110%
         if (observe-object.get-area() < 1.10 * area)
            move(u, v, FORWARD);
         else begin
            depth-mode := NO-DEPTH;
            move(u, v, 0);
         end;
      else if (depth-mode == MOVE-AWAY)
         // move backward until the area shrinks to 90%
         if (observe-object.get-area() > 0.90 * area)
            move(u, v, BACKWARD);
         else begin
            depth-mode := NO-DEPTH;
            move(u, v, 0);
         end;
   end;
end.
```

References

[1] P. Agre, D. Chapman "What Are Plans For?" in *Designing Autonomous Agents - Theory and Practice from Biology and Back*, edited by Pattie Maes, MIT Press, 1990.

[2] J. Albus, H. McCain, R. Lumia "Nasa/nbs standard reference model for telerobot control system architecture (nasrem)". Technical Report NISTTN 1235, National Institute of Standards and Technology, 1989.

[3] Y. Aloimonos, Z. Duric, C. Fermueller, L. Huang, E.Rivlin, R. Sharma "Behavioral Visual Motion Analysis" in *Proceedings of the Image Understanding Workshop*, January, 1992.

[4] R. Arkin "Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation" in *Designing Autonomous Agents - Theory and Practice from Biology and Back*, edited by Pattie Maes, MIT Press, 1990.

[5] P. Bonasso and D. Miller "Instantiating Real-World Agents", Technical Report, AAAI Fall Symposium, 1993.

[6] R. Brooks Elephants don't play chess, *AI Journal*, 1990.

[7] R. Brooks Behavior Language Manual, MIT Technical Report, 1992.

[8] D. Chapman *Vision, Instruction, and Action*, MIT Press, 1992.

[9] R. Cohen and D. Akin "The Role of Supplemental Camera Views in Space Teleoperation" in *Proc. Teleoperation '93*, 1993.

[10] R. Kohout, D. Musliner, and J. Hendler "Dynamic Reaction on the Maruti Hard Real-time Operating System, Technical Report, UMCP (in press).

[11] E. Gat "ALFA: A Language for Programming Robotic Control Systems" in *Proceedings of the IEEE Conference on Robotics and Automation*, May, 1991.

[12] I. Horswill "Polly: A Vision-Based Artificial Agent" in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, July, 1993.

[13] L. Kaelbling, S. Rosenschein "Action And Planning in Embedded Agents" in *Designing Autonomous Agents - Theory and Practice from Biology and Back*, edited by Pattie Maes, MIT Press, 1990.

[14] D. Miller, R. Desai, E. Gat, R. Ivlev, J. Loch "Reactive Navigation through Rough Terrain: Experimental Results" in *Proceedings of the AAAI National Conference on Artificial Intelligence*, July, 1992.

[15] V.S. Ramachandran "Interactions Between Motion, Depth, Color and Form: The Utilitarian Theory of Perception" in *Vision Coding and Efficiency*, edited by Colin Blakemore, Cambridge University Press, 1990.

[16] J. Sanborn and J. Hendler "A model of reaction for planning in dynamic domains" *International Journal of AI and Engineering,* Volume 3(2), April, 1988.

[17] R. Simmons "Structured control for autonomous robots" in *IEEE Transactions on Robotics and Automation*, 10(1), February, 1994.

[18] Spector, L. and Hendler, J. "Planning and Control across Supervenient Levels of Representation," *I.J. Intelligent and Cooperative Information Systems*, 1(3-4), December, 1992

[19] Spector, L. "Supervenience in Dynamic World Planning" Doctoral Dissertation, University of Maryland, 1992.