Featured Researcher

Fault Tolerant Behavior-Based Robots

Laura Wong, 2008

Advised by William Wong, MS
Department of Mechanical Engineering

What is robotics research all about?
Robotics research has a wide variety of topics—from medical robotics to toy robotics. My main research involves search and rescue robotics using sensors and fault tolerance. The military and many security companies use search and rescue robotics to save and preserve lives. For example, if there is a suspicious cave, one could send in a robot to search the cave for any hazardous or dangerous object/people instead of sending in a human. If it turns out that there is indeed something hazardous in the cave, they will only lose a robot, which can be remade, and not someone's life. Robotics research is widely used to help humans in many ways.

Who or what motivated you to start on this research project?
I began working in the field of robotics the summer before my freshman year in high school. I became interested in robotics a few years prior to that, when my siblings were playing with a toy robot they received as a gift. With the addition of my father working with computers, I began to develop interest in computer programming. With the support and motivation of my father and the rest of my family, I stuck with robotics throughout my high school career. Throughout the years I gained knowledge of different sensors used for robotics. For my final project, I incorporated everything from the mechanics aspect to the computer programming aspect, which I learned over the years.

While doing this research project, what do you think was your biggest obstacle and how did you overcome it?
This particular research project took about 9 months to be completed. Over the course of 9 months, I encountered some minor obstacles such as power supply and wiring confusion. Each robot ran off of 8 separate C batteries, but the problem was that they would only run about 10 minutes at a time. Aside from that, there are many different wires in each robot that connect everything together and if one became loose it was a slight challenge to figure out what came undone. The main obstacle I had to tackle, however, was to learn the programming language of JAVA, which was completely new to me. Over the years I have worked with BASIC, C++, and Pro-Log. Working with a variety of these programming languages can be quite confusing when you are writing a program due to the fact that many of them are closely related to each other. But with a lot of determination and even more patience I was able to grasp JAVA.

After completing your research project, what do you think was your most fulfilling experience?
After completing this research project I participated in a national science fair held in Mercer County, New Jersey, where I received the Grand Award. By winning the Grand Award, I was able to participate - for my third consecutive year - in Intel’s International Science and Engineering Fair, held in Portland. There I participated in a weeklong science fair where I was able to present my project to a variety of judges and to the public. I received second place in the Computer Science group and a variety of special awards. The main award I was most proud of was the top award from the United States Army, which included a medal, $3,000 in savings bonds, and a two-week trip to London where I participated in an international science forum with 250 young scientists from all over the world.

Any advice you can give to fellow undergraduates who would like to do this kind of research?
The main advice I can give to fellow undergraduates is to not get discouraged when something doesn't work the first, second, or third time. Robotics is a very complex area and you need to start at the basics to get a slight grasp on it. It would also be helpful to find a mentor who knows a lot about the area. Having a mentor over the years made my experience more valuable and allowed me to get used to the field of robotics more easily. Finally, allow your family and friends to take part in the experience - having support from many people makes the experience easier and gives you boost to continue on with your research, especially when you are faced with obstacles.
warm robots, a collective of cooperating robots, provide a way to apply many devices to a problem. Utilizing many robots brings a level of redundancy that improves overall fault tolerance — where a system continues to run possibly in a degrade mode when a fault occurs. They can also provide mutual application support such as providing positional references to each other, as well as increasing the range of the overall system compared to a single robot.

Creating software that addresses fault tolerance can be difficult, but one way to simplify this task is to use meta-behaviors within a hardware system designed with sensors and resources with overlapping capabilities.

A variety of behavior-based programming methods have been used over the years. The robust nature of behaviors allows behavior-based robots to handle the inaccuracies of sensors and resources. Plan-based solutions that try to use techniques like dead reckoning and accurate maps tend to run into accuracy problems the longer they run.

Behavior-based programs are usually easy to create and modify, although the techniques for designing and testing tend to be different from many other programming tasks because of the way behaviors interact. Behaviors are invoked dynamically so the interaction between behaviors, which compete for system resources, is often not explicitly stated in the design making testing. This is a key part of a system's implementation.

Behavior-based systems can be implemented on very small computing platforms such as the Mbot used in this paper and shown in Figure 1. Behaviors tend to be relatively simple requiring minimal storage. Systems that exhibit complex actions can be created with less than a dozen behaviors.

Behaviors typically use a set of inputs to control how resources are used. For example, an obstacle avoidance behavior may stop the wheel motors of a robot when the input from a touch sensor indicates the robot has collided with an obstacle. A different behavior may cause the wheels to move forward if there is no obstacle. Yet another may cause the wheels to turn the robot towards a goal such as a light source.

One technique for determining which behavior gains control of a resource is called subsumption. In this case, behaviors can inhibit inputs or suppress outputs of other behaviors. Another technique is called motor schemas, where weighted outputs from behaviors are combined to control a resource.

Typically, the behavior with the highest weight has the greatest impact on a resource's action. Behavior prioritization is yet another technique where priority can be implemented in a number of ways, such as keeping behaviors in priority order and using the first behavior on the list that wants to use a resource.

Many behavior-based systems determine what behaviors to use by repeatedly scanning a list of active behaviors. The cycle time determines how quickly the system, in this case a robot, responds to changes in the system's environment. This approach is used in this paper. Minimizing the cycle time improves the system's responsiveness.

Meta-behaviors fit into this cycle-style approach very easily. The key difference between a behavior and a meta-behavior is the kinds of inputs that are used and the actions performed by the behaviors. For our purpose, a behavior uses system inputs and manipulates system resources while meta-behaviors use information about system inputs and resources to manipulate behaviors.

Figure 2 shows how a typical fault-tolerant meta-behavior operates along with its main components. In general, a meta-behavior contains a list of normal and error behaviors. When a fault is detected, it removes any normal behaviors and replaces them with behaviors that will address the problem. The process is reversed if the fault is corrected. The meta-behavior may initiate other corrective action when a fault is detected allowing the fault to be corrected.

The figure uses Universal Modeling Language (UML) diagrams to describe this process. UML is used in other
diagrams such as state charts and sequence diagrams within the paper. The explanations within the paper should allow anyone unfamiliar with UML to understand the diagrams.

Fault tolerance has been implemented in other fashions such as creating behaviors that explicitly take faults into account. This approach works but it can lead to more complex behaviors. It also tends to use many active behaviors whose overhead can impact system performance.

Fault tolerance is easier to achieve if there is some level of redundancy in the hardware. In a swarm, numerous instances provide redundancy—assuming the loss of a robot is acceptable. Replication of a particular subsystem, or providing sensors and resources with overlapping capabilities, is another way to provide redundancy within a robot. The latter was chosen as a way to make the Mbot a more robust system. It also made it easier to initially test the meta-behavior approach using a single robot when one of the subsystems was disabled.

Figure 3 and Table 1 show the complement of sensors and resources employed in the Mbot. The Mbot uses a Video Grid Sensor (VGS). The VGS is an intelligent camera I developed. It uses color information to locate obstacles and identify objects by color. This allows other robots to be color coded for identification.

The Mbot can continue functioning even if one of these resources is disabled, although it may not be able to perform useful work if more than one subsystem fails. Another subsystem not listed is the wheel motor control. If these fail, the robot cannot move, but it could still provide a readily identifiable landmark to other robots within a swarm. Meta-behaviors and error behaviors can be created to make this happen.

Materials and Methods

This section addresses hardware and software architecture employed, the test environment, and the types of tests used to exercise the architecture. As with most robotic experiments, the hardware, software, and environment are restricted to minimize overall complexity. This is especially true for systems with limited resources.

Robot Construction

The Mbot is a custom robot. The main hardware subsystems are shown in Figure 4. The Java-based, Javelin microcontroller is the centerpiece, although there are a number of other microcontrollers within the system. These include the ServoPal, which is used for servo control, and the video grid sensor (VGS).

The Mbot has a pair of drive wheels. It can pivot on its center axis. Wheel encoders make more accurate movement possible in addition to providing odometer information.

Obstacle avoidance is handled by the VGS and the touch sensor skirt (Figure 5) that surrounds the Mbot. The VGS only provides information about objects in front of the Mbot, although it can tilt and pivot the camera providing a wider and longer range of vision compared to a fixed camera. The VGS provides simple image analysis designed to deliver a compact image map to the Javelin via a serial port.

<table>
<thead>
<tr>
<th>Name</th>
<th>Communication</th>
<th>Range</th>
<th>Orientation</th>
<th>Angular Coverage</th>
<th>Obstacle Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio</td>
<td>Yes</td>
<td>20 m</td>
<td>No</td>
<td>N/A</td>
<td>No</td>
</tr>
<tr>
<td>IR Beacon</td>
<td>Yes</td>
<td>2 m</td>
<td>Yes</td>
<td>360°</td>
<td>IR beacon</td>
</tr>
<tr>
<td>Video Grid Sensor</td>
<td>No</td>
<td>1 m</td>
<td>Yes</td>
<td>90°</td>
<td>Yes</td>
</tr>
<tr>
<td>Touch Sensor</td>
<td>No</td>
<td>0.002 m</td>
<td>Limited</td>
<td>360°</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Mbot Device Input And Communication Capabilities
The IR beacon shown in Figure 6 provides limited orientation information allowing the Mbot to locate another beacon within 45°. It can also exchange serial information with another beacon at low speeds. The radio transceiver also provides a non-line of sight communication link to other robots. It is faster and more accurate than the IR beacon communication.

The Javelin has a number of limitations that significantly impacted the design of the system. First, it only has 32Kbytes of memory for both program and data. Using intelligent peripherals like the ServoPAL and VGS reduces the size of behaviors and supporting program code. Second, the Javelin has only sixteen I/O pins. The intelligent devices make better use of the pins and a multiplexor chip helps cut down the number of inputs required.

Memory was not the only software limitation. The Javelin implements peripheral interfaces using virtual peripherals that are allocated from memory. There is a limit of six active virtual peripherals (VP). The timer service uses one of the six active VP slots. Table 2 shows the list of VPs used in the system.

The total number of VPs exceeds the five available active VPs so it was necessary to activate and deactivate VPs as needed. This tended to slow down the operation of the robot because it was often necessary to stop movement or other actions while active devices where changed and used.

**Software Architecture**

The system employs a single tasking, priority-based arbiter to support the behavior-based system because the Javelin does not support multitasking or garbage collection. Three list managers keep track of the three major items: inputs, behaviors, and resources. Input objects provide status information about the Mbot and its environment. The behavior list contains all active behaviors and meta-behaviors. Resources are controllable subsystems. The primary resource is the wheel servos.

The first thing the arbiter does is check all inputs. Each input should store its current information so all behaviors that test an input will see the same information. Changing this information on the fly instead will cause all sorts of problems. A behavior may check any number of inputs. An input object may provide a range of information.

If a behavior meets its internal conditions it will then try to acquire any necessary resources. The highest priority behavior will be allowed to use a resource. The arbiter checks all resources after it checks the behavior; a resource will notify a behavior if it can use it. It will first tell any behavior that may currently be using the resource if a higher priority resource needs it. This allows a higher priority behavior, such as, avoiding a collision, to take control of the robot's movement.

Figure 9 shows a simple behavior in a simplified diagram. The Arbiter/Managers column condenses down the arbiter and list manager interaction of Figure 8 into a single transaction. The MoveForwardBehavior causes the robot to move forward regardless of whether an obstacle is in front of the robot. This allows another behavior to check for obstacles. The robot starts moving forward the first time the WheelsResource is acquired.

The behavior in Figure 10 shows a more complex ballistic interaction. The detection of an obstacle starts a movement sequence in which the robot stops, backs up, and then pivots...
to one side so it is no longer facing the obstacle. It then releases the wheel resource allowing another, lower priority behavior like the one shown in Figure 9 to take control. If there is still an obstacle in the way then the behavior in Figure 10 will gain control and the process will be repeated.

Test Environment

Since the Mbot has limited mobility and the VGS has limited accuracy, it was necessary to control the test environment. The VGS is also sensitive to some lighting conditions. For example, bright sunlight will overwhelm the camera and incandescent lights emit IR radiation that can limit its accuracy.

The test environments consisted of rooms with an open area of 4m x 5m. The floor was flat with tiles or low pile carpet. The VGS was able to handle variegated floor colors. The light was fluorescent with minimal shadows. The skirts of the robots were color coded so they could be identified using the VGS. This allowed an Mbot’s VGS to recognize another Mbot even if the IR beacon failed.

Objects had a different color than the floor and the Mbots. They were tall enough to come in contact with the touch sensor skirt. Objects were at least 0.5m apart if the Mbot was expected to go between the objects. This distance was sufficient for the Mbot to move through the area and for the VGS to recognize the opening.

Tests

Two major goals were used to drive the creation of the behaviors. The first was to Find and Face another robot. The second was to Follow the Leader robot. Figure 12 shows how this occurs and the behaviors and actions performed. The follower robot wanders until it finds the leader. It needs to be within range of the leader to recognize it. The follower was placed near the leader so it did not have to wander too far. This was done to minimize the time needed to complete an experiment.

The first step was to create a set of behaviors to allow the Mbot to handle each task under normal operating conditions. The second step was to develop meta-behaviors that could detect errors in subsystems. The third step was to generate error behaviors that operate with a subset of inputs and resources. Finally, the normal, error, and meta-behaviors were combined and tested.

The combined behaviors were tested by simulating a fault. For example, a VGS fault was simulated in software. It was also simulated by disconnecting its serial communication link and by covering the camera lens.

Results

Behaviors and meta-behaviors created were sufficient to perform the tasks presented. Development of error behaviors proved to be straightforward since it was possible to test a small subset of behaviors. For example, a more advanced approach to find the leader uses the VGS to stay a short distance away. A simpler approach without the VGS is to use the touch sensors. In this case, the robots came in contact. A third approach is shown in Figure 13. In this case, meta-behaviors to handle a follower robot VGS fault are found in both robots. The meta-behaviors and error behaviors are different for both robots but they do cooperate. The overall system performance is lower
than when the follower robot VGS is working but the results are still useful.

In the case of the leader robot, the meta-behavior waits for a message from a follower robot. It then added a set of error behaviors that face the follower and communicate with it so that the leader’s VGS can be used to determine when the follower is close enough. The process significantly slows down system operation because the leader had to face the follower then later turn away to move forward and finally turning back again. Still, writing the error behaviors needed to perform this task was a straightforward process.

The meta-behavior on the follower had both normal and error behaviors. Testing was easy because the error behaviors for the leader and follower could be loaded and tested independently of the normal behaviors. In many cases, the set of error behaviors included normal behaviors created for other tasks.

System overhead for meta-behaviors did not add too much overhead because the input tests were usually simple go or no-go decisions. Simple subsystem self-test operations were fast. One difficulty was trying to determine that the VGS was not working when the lens was covered because the condition was similar to having the robot in an open area. Two solutions to detecting this fault seemed to work well. The first was to indicate a fault if the front touch sensor indicated an obstacle and if the VGS had not detected any obstacle since it was started. The second was to indicate a fault if an initial self-test tilted the VGS up so it could scan the horizon after the test took more time than the typical sensor testing process.

Unfortunately the touch sensor design was prone to needing readjustment. The sensor worked well once the four sensor contacts were checked, but they could easily be altered if the robot was picked up and moved from one site or position to another. A different design will be considered in the future.

Most of the testing was a success. The robots successfully completed each task assigned to them. The creation of the behaviors necessary to perform the task was relatively simple compared to the overall system design. A good deal of time was spent on refining the arbiter, inputs, and resources. Memory constraints limited the complexity and number of behaviors that could be loaded at one time, although some space could have been conserved by removing debugging information. Still, the system implementation was sufficient to prove the concept.

The system was able to handle a dozen behaviors, but when they were doubled, the system ran into more memory constraint issues than performance issues. Still, the overhead for checking this many behaviors was high. Certain programming concessions were made, such as using the ServoPAL to run the wheels to cover a fixed distance instead of waiting until an obstacle was detected. It prevented the robot from going too far if the arbiter cycle time ran too long. This could sometimes occur when a meta-behavior was checking an input for a fault and the test took more time than the typical sensor testing process.

One problem encountered in previous projects was the speed and performance of the robot. This was not a problem this time, except when a fault was detected and a lower performance was expected in this instance. The ability of the robots to move and search at the same time was primarily due to the use of ServoPAL and VGS coprocessor. Handling the VGS interaction asynchronously also helped. This was done by initiating a VGS operation immediately after the VGS results were obtained. The VGS processing occurred concurrently with the rest of the arbiter cycle.

Meta-behaviors did reduce the number of behaviors that were active at one time. It also simplified the creation of error behaviors. Eliminating the meta-behaviors and combining and enhancing the normal and error behaviors so they handled faults more complex than anticipated, resulting in more active and larger behaviors. Performance was no better and often slower than the meta-behavior approach.

As noted in the Results section, testing provide to be easier using meta-behaviors. Testing a smaller number of combinations is almost always easier than testing a larger number of combinations.

There was not sufficient time or memory capacity to attempt to use meta-behaviors for planning or to implement high level plans created externally, but the results from using
this approach for fault handling indicates that the approach would be useful. It is an area worth investigating.

Other issues arose as the number of behaviors increased and the tasks became more complex. First was the concept of a meta-input and a generic fault meta-behavior class. A standard fault meta-behavior object would have a standard interface to a meta-input object to test for a fault. Therefore adding most fault support would consist of creating a subclass of meta-input for a particular input or resource, and then filling in the normal and error behavior lists of a standard fault meta-object. Second, there was no way to track what behaviors were actually removed when a meta-behavior switched between fault conditions. This could result in restoring a behavior that was not previously removed because it was removed by another meta-behavior. An approach to solve or at least identify this condition will be found in the future. Third, the use of changeable, numeric priorities for behaviors eliminated the need for dealing with sorted priority schemes and allowed resources to be managed independently when necessary. For example, one behavior may be using the radio resource while another is using the wheels. It provides a sort of multitasking within behaviors.

Another area of study would be the use of multitasking arbiters and multiple arbiters in a more complex system that has subsystems with minimal interaction. For example, a robot with two arms may work better with three arbiters for the robot movement and each arm with meta-behaviors coordinating the interaction between subsystems.

Meta-behaviors can easily be implemented as standard behaviors but formalizing the process makes it easier to deal with and allows the reuse of common services, like adding and removing the lists of behaviors when a fault occurs. They definitely made programming the Mbots easier.