Property Mapping: a simple technique for mobile robot programming

Illah R. Nourbakhsh
The Robotics Institute, Carnegie Mellon University
Pittsburgh, PA 15213
illah@ri.cmu.edu

Abstract

The mobile robot programming problem is a software engineering challenge that is not easily conquered using contemporary software engineering best practices. We propose *robot observability* as a measure of the diagnostic transparency of a situated robot program, then describe *property mapping* as a simple, language-independent approach to implementing reliable robot programs by maximizing robot observability. Examples from real-world, working robots are given in Lisp and Java.

Introduction

The recent availability of inexpensive, reliable robot chassis (e.g. ActivMedia Pioneer, IS-Robotics Magellan, Nomad Scout) has broadened the accessibility of mobile robotics research. Because these robots consist of fixed hardware out-of-the-box, this technology is shifting the emphasis of mobile robot research from a joint hardware and software design process toward hardware-unaware mobile robot *programming.*

This is a software design problem, and yet software engineering best practices are not very helpful. Mobile robots suffer from uncertainty in sensing, unreliability in action, real-time environmental interactions and almost non-deterministic world behavior.

New programming languages and architectures have been born out of projects to create reliable, real-world robots (Bonasso and Kortenkamp, 1996), (Brooks, 1986), (Firby, 1987), (Horswill, 1999), (Simmons, 1994). Several have demonstrated their efficacy in working mobile robots (Horswill, 1993), (Krotkov et al., 1996).

These existing solutions all effectively constrain the programmer during the process of robot programming, but none provide satisfactory guidance regarding the incremental process of constructing and debugging robot programs. We suggest that one reason the best implementations in these languages succeed is because they facilitate robot debugging. Even more so than with non-robotics systems, diagnosis and subsequent modification of robot software is the chief time sink of robot programming. We believe there is a single key requirement for making robot programming tractable: maximize run-time diagnostic transparency.

In this paper we present *robot observability* as a predictor for diagnostic transparency. Then, we present a language-independent technique called *property mapping* for constructing robot programs that are diagnostically transparent and thereby achieve high degrees of reliability.

The Robot Programming Problem

A mobile robot is a situated automata, or a module that comprises one part of a closed system together with the environment. Fig. 1 shows the standard depiction of a robot, with its outputs labeled as *actions* and the outputs of the environment in turn labeled as *percepts*.

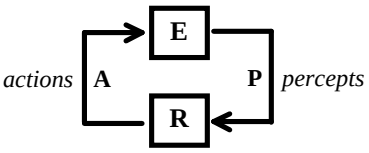

Figure 1: The standard view of an embedded robot

This view can be confusing, as the module *R* is not meant to depict the physical robot, but rather the automata implemented by the *robot algorithm* (Halperin, Kavraki and Latombe, 1998). The physical robot is a component of the environment, and it is the interface through which the robot automata receives percepts and acts

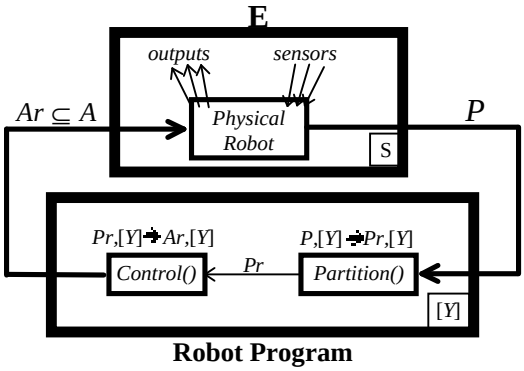

Figure 2: A detailed view of the robot-environment system

In Fig. 2, we depict the standard view in greater detail. Without loss of generality, we conceptually decompose the automata, or robot program, into a perceptual partitioning module and a control module. *Partition*() is

simply a function that may implement perceptual abstraction by mapping various percepts in *P* to the same abstract percept in *Pr*. *Control*() is a Moore automata, or a function from the perceptual input (and possibly internal state) to an output (and possibly a new internal state). We use the set *Ar*, which is a subset of *A*, to denote the range of *Control*().

The robot programming problem can be posed as follows. The mobile robot software designer is given an environment **E** with state set *S*, including a physical robot with fixed sensors, outputs, a set of possible sensor value-vectors *P* and a set of possible output vectors *A*. The designer creates the Robot Program, and in so doing chooses the effective output range *Ar*, the perceptual partitions *Pr* and the internal state set *Y*, together with the mappings $P,Y \rightarrow Pr,Y$; $Pr,Y \rightarrow Ar,Y$.

Sensor noise is a significant problem in mobile robot programming, and so the separation of the robot program into a perceptual partition followed by state and action update follows the natural course of most robot programming solutions. This is particularly the case in functional programs where state is not used and therefore the robot program simply maps the current perceptual input to an output: $P \rightarrow Pr \rightarrow Ar$.

Fig. 3 shows a working example of a functional program. This program is designed for a differential-drive robot with a sixteen-sonar radial ring. It indefinitely servoes the robot to face the closest object. If one walks around the robot, the robot spins in place, attempting to face the person.

```
Public void turnClosest
  (java.awt.event.MouseEvent event)
  {
   int shortestDirection; int speed;
   boolean flag = true;
   RC.turnSonarsOn(); RC.GetState();
   while (flag) {
    RC.GetState();
    shortestDirection=calcShortestDirection() -1;
    if (shortestDirection > 8)
     shortestDirection -= 16;
    if (Math.abs(shortestDirection) > 2)
     RC.setVel(-(shortestDirection*40),
               (shortestDirection*40));
    else RC.setVel(-(shortestDirection*20),
                   (shortestDirection*20));
    } } // turnClosest() //

  int calcShortestDirection()
  {
    int minVal = 255; int minIndex = 0;
    for (int i=1; i<17; i++)
      if (RC.stateArray[i] < minVal) {
         minVal = RC.stateArray[i]; minIndex = i;}
    return minIndex;
  } // calcShortestDirection() //
```

Figure 3: Java code for the Turn-Closest program

This code shows the separation of the software into a perceptual reduction, "what direction is the closest object?" in `calcShortestDirection()` followed by action selection, "which way should I turn and how hard?" in the remainder of the `turnClosest()` method.

When students of *Mobile Robot Programming*[1] do the `turnClosest` assignment on their Nomad Scout robots, the two most common reasons for poor behavior correspond to errors in the perceptual partition step and the control step. In the former case, the $P \rightarrow Pr$ mapping fails because students use algorithms that are too complex in determining the location of the closest object (e.g. minimum of adjacent *n* sonar readings). In the latter case, the robot successfully determines the direction to the closest object but the speeds chosen in $Pr \rightarrow Ar$ are too high and so the robot overshoots and oscillates (particularly when the student is unfamiliar with control theory).

The challenge of robot diagnosis is that the same poor robot behavior may be caused by either a bug in *Partition*() or a different bug in *Control*(). Diagnosis would be easier if the robot program were written in a way that enables the history of values for the internal variables of the Robot Program to be observable. In the case of a state-less, functional program, the only internal variable is the abstract percept. We denote the value of the abstract percept over time as *Pr**.

Robot Observability

State observability is defined as the ability of a robot to acquire the value of the state *s* of the Environment E (see Fig. 2). A robot in a fully observable world can acquire the trajectory of environmental state values over time: *S**.

We define the term, *robot observability*, as an analogue to state observability, where the target is not the Environment but its companion, the Robot Program.

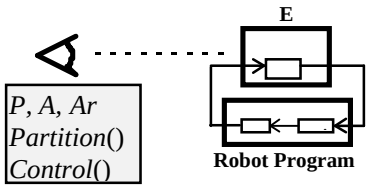

Figure 4: The addition of an observer to the robot system

Suppose that an observer is added to the closed system of Figures 1 and 2. This observer is given a description of the sets *P* and *A* as well as full knowledge of the static robot program, including *Partition*() and *Control*() (See Fig. 4). During run-time, the observer only directly sees the Environment module, but perceives changes in E only to a finite level of acuity.

We define *robot observability* as the degree to which the observer can identify the trajectory of the internal values of the Robot Program, *Pr** and *Y**.

A robot program that is fully observable has transparent internal values. Of course, this depends not only on the structure of the robot program but also on the

[1] CS224l at Stanford University; 16x62 at Carnegie Mellon University; CS1699 at University of Pittsburgh.

observability of the Environment. Consider a caterpillar robot with a single forward-facing sonar sensor. The robot program partitions the set of possible sonar values into two possible values that comprise *Pr*: {*close, far*}. The *Control*() algorithm is functional, simply mapping *close* to zero motion and *far* to positive speed. Lisp code for the robot program is shown in Figure 5.

```
(defun caterpillar-control ()
  (loop
    (r-move (if (close-obstacle) 0
                *forward-speed*) 0)))
(defun close-obstacle ()
  (< (get-sonar) 6))
```

Figure 5: Code for the caterpillar robot program.

This system has full robot observability for any observer who can reliably discriminate the robot's motion.

It can be shown that for any functional robot program with a bijective mapping from *Pr* to *Ar*, the robot is fully observable if the actions in *Ar* can be discriminated by the observer. This may explain the success of functional and quasi-functional (e.g. reactive) programming in robotics, because viewing the actions of the robot can often provide a direct window into the percept stream arriving at the robot. When the robot misbehaves, the designer can easily determine if the percept violates implicit assumptions made by the designer, or if the functional mapping from percept to action needs to be modified.

Because the composition of *Ar* is an important design choice for the robot software engineer, we suggest a heuristic that is applicable whether the engineer is designing a functional, reactive or state-based robot program: *prefer robot program solutions with greater robot observability*.

For example, consider a mobile robot that avoids obstacles and must reach a goal location. Fig. 6 shows top-level Lisp code for a synchro-drive robot chassis.

```
(defun swervebot-control ()
  (loop (r-move 10
           (cond ((close-obstacle-right) 30)
                 ((close-obstacle-left) -30)
                 ((goal-way-right) 30)
                 ((goal-way-left) -30)
                 (t 0)))))
```

Figure 6: Code for avoiding an obstacle and achieving a goal

The code in Fig. 6 demonstrates a case-based approach to specifying rotational velocities for the robot. Four most basic cases are identified, and each is assigned a discrete rotational speed and direction. A popular alternative to this case-based approach is a continuous-valued approach, in which the speed is a continuous function of the sensor values (Borenstein and Koren, 1991). For instance, the rotational speed in the case of `goal-way-right` could be a function of the number of degrees disparity between the heading to the goal and the robot's current heading.

By now it may be apparent to the reader that Fig. 6 contains a simple bug. The signs of the rotation velocities are incorrect in the case of the third and fourth condition; the robot will swerve away from obstacles but also away from the goal when there are no obstacles. This error was not contrived but, rather, it was made by the author on his first try at the robot program.

The behavior that resulted during first testing on a Nomad 150 robot demonstrates the difficulty of debugging mobile robot code. During testing, the robot would, at times, turn away from the goal. Were the sonar sensors seeing phantom obstacles, or was there an error in the rotational encoder? The robot appeared to work correctly at other times, because the designer was standing next to the robot. Unbeknownst to the designer, the robot was executing, not the `goal-way-left` case (which would have caused a rotation in the "wrong" direction), but the `close-obstacle-right` (which seemingly turned towards the goal) because its sensors were picking up the designer.

This debugging difficulty stems from a lack of robot observability: when the robot swerves left, it cannot be determined by observation whether it is doing so because of `cond` case 1 or (in the corrected code) `cond` case 4. The solution, in keeping with the heuristic to maximize robot observability, is to populate *Ar* with actions from *A* that are recognizably different for each of the four different abstract percepts in *Pr*, as shown in Fig. 7.

```
(defun swervebot-control ()
  (loop (r-move 10
           (cond ((close-obstacle-right) 30)
                 ((close-obstacle-left) -30)
                 ((goal-way-right) -15)
                 ((goal-way-left) 15)
                 (t 0)))))
```

Figure 7: Improved and debugged swervebot code maximizing robot observability

When it is possible to design a robot programming solution with only a handful of cases, this technique may be advantageous relative to the continuous-valued programming approach due to its high robot observability.

This bias toward a small number of cases is just one side effect of the heuristic to maximize robot observability. The general effect of maximizing robot observability is a bias toward reducing the size of *Ar* maximally, and ideally until the action space is no larger than the robot program's internal feature space: $|Ar| = |Pr \times Y|$. An overly large set of actions in *Ar* will fail to be discernible by the observer.

In constructing *Ar*, the designer is performing *action sampling*, in which a subspace of the available action space is being delineated for use by the robot. The perceptual complement is *percept abstraction*, in which the percepts of the robot, *P*, are partitioned into $|Pr|$ values. If the number of partitions selected is too small, then the robot may fail to recover relevant information from the environment. If $|Pr|$ is too large, then the robot may be discriminating detail that is irrelevant to its task.

In summary, an approach to maximizing robot observability is to minimize the size of *Pr* and *Ar*, while including sufficient information in *Pr* to capture relevant

environmental detail and making *Ar* inclusive enough to maintain the robot's expressiveness. Next, we propose an incremental method for constructing a robot program along these lines.

Property Mapping

A property is a subset of the set of environment states, *S*. Intuitively, a property captures the value of one feature of the world while allowing other features of the world to range freely. For example, property *p1* may correspond to the robot faced by an obstacle on its left and no obstacle on its right. This property would denote every such state, varying the robot's absolute orientation and the existence of obstacles behind the robot freely, for example.

In the case of robot programming, our intent is to use properties to capture the aspects of environment state that are most relevant to the programming problem. For example, in the case of a mobile robot destined to serve hors d'oeuvres, the designer will care about properties corresponding to the location of obstacles, hungry persons and the status of the tray: {*obstacle near, obstacle far, hungry person near, hungry person far, tray has cookies, tray is empty*}.

When designers think in property space, they can choose to define the set of properties Pr and then the output function g without considering the observability of the properties using the robot's actual sensors. Surprisingly, ignoring the sensory shortcomings of the robot during part of design-time can be beneficial. We propose such a method for robot program design:

1. Suppose that *S* is fully observable. Select a new property that can map to a single, coherent action.
2. Select a specific action for this property. Avoid choosing the same action for two distinct properties.
3. Implement the property-action pair in code. Test if possible.
4. Unless finished go-to Step 1.

By first ignoring the partial observability problem, the designer avoids simultaneously solving both the action selection problem and the perceptual/state update problems. Instead, the designer creates a set of property-action pairs, where only the most relevant properties are introduced. Since we assume full observability, this approach generates all of the key actions that belong in *Ar* regardless of the robot's particular perceptual limitations. In Step 3, the designer encodes recognition of the chosen properties into the robot program, thereby addressing the perception/state update problem on a property-by-property basis.

Solving the problem incrementally first minimizes the size of *Ar* then introduces only as many perceptual partitions into *Pr* and only as much state *Y* as is necessary to detect the target properties.

Consider, for example, the robot programming problem for Cheshm, a mobile robot that avoids obstacles using a solitary depth-from-focus sensor. Cheshm's depth-from-focus sensor provides just three levels of range information (i.e *close, medium, far*) in a 3 x 5 grid. The programming challenge is to design a robot program that would actively wander while avoiding both convex and concave (e.g. stairs) obstacles in order to demonstrate the sensor's robustness.

The first property chosen, called `danger-closep`, recognizes the existence of any convex obstacle dangerously close to the robot. The resulting control program, shown in Fig. 8, made Cheshm a caterpillar robot.

```
(defun cheshm-control ()
  (loop (r-move (if (danger-closep) 0 80) 0)))
```

Figure 8: Preliminary Cheshm control code.

We tested this first version of Cheshm by running the robot in a variety of circumstances, stepping in front of the robot and placing various objects in its way to see if it would react. After debugging the sensing system, additional property-action pairs were added to implement swerving around obstacles based on their positions: {*danger-leftp, danger-rightp, medium-leftp, medium-rightp*}. Fig. 9 shows the complete action control code for Cheshm.

```
(defun cheshm-control ()
 (loop (if (concave-obstaclep) (turn-around)
        (progn
          (setf translation-velocity
             (cond ((danger-closep) 0)
                   ((medium-closep) 40)
                   (t 80)))
          (setf rotation-velocity
             (cond ((danger-leftp) -40)
                   ((danger-rightp) 40)
                   ((medium-leftp) -20)
                   ((medium-rightp) 20)
                   (t 0)))
          (r-move translation-velocity
                  rotation-velocity)))))
```

Figure 9: Cheshm's actual action-selection code

The rotational speeds for medium and close obstacles differ significantly (`20` versus `40`) so that to the designer/observer can easily recognize the triggering property. An important goal of the Cheshm project was to demonstrate that vision can safely detect concave obstacles. Note that `concave-obstaclep` uses a different action from those used below it for convex obstacles. This enabled us to observe Cheshm's ability to explicitly recognize concave obstacles, as it turns in place 180 degrees upon encountering stairs and ledges (Nourbakhsh et al., 1997).

A more recent working example is T.A.W. (Texas Alien Woman), a contestant in the 1999 AAAI Hors D'oeuvres competition. In this competition, robots search for humans in the room, offer them appetizers, and return to their refill stations for more appetizers when they run out. T.A.W. was an entry designed to demonstrate the use of inexpensive pyroelectric sensors to detect its human

targets. The method, `gohome()`, was written to control T.A.W. during its return to the refill station once it had run out of cookies. The top-level goal for `gohome()` is to achieve a specific position, the home base, while avoiding unpredictable, moving obstacles along the way. This code (Fig. 10) was written using the property-mapping technique, starting with the property, `closeObstacle`.

```
public void gohome() {
 int forwardVel = 0; int goalDist = 0;
 int rotationVel = 0; double rotationGoal;
 int rotIntGoal; int currentRot;
 rc.GetState();
 goalDist = Math.abs(rc.stateArray[rc.XPOS]) +
            Math.abs(rc.stateArray[rc.YPOS]);
 rotationGoal =
   Math.atan2((rc.stateArray[rc.YPOS]),
              (rc.stateArray[rc.XPOS]));
 currentRot = rc.stateArray[rc.THETA];
 rotIntGoal = (int)(3600.0 * rotationGoal /
 if (rotIntGoal < 0) rotIntGoal += 3600;
 if ((personp()) && (canExcuseMe())) {
   resetExcuseMeTimer();
   System.out.println("I must go get cookies
                       excuse me");
   myTongue.playSound("out.au");
 } // if
 if (goalDist < homeThreshold) {
   move(0,0);
   myState = REFILLING;
   //System.out.println("exitingtoREFILLING");
 } else if (closeObstacle()) {
   //System.out.println("Close obstacle... ");
   forwardVel = 0;
   rotationVel =
     computeRotCloseObstacle(currentRot,
                             rotIntGoal);
 } else {
   //System.out.println("no obstacle...");
   forwardVel = computeForVelTo(goalDist);
   rotationVel = computeRotTo(currentRot,
                              rotIntGoal);
 }
 move(forwardVel, -rotationVel);
} // gohome()

int computeForVelTo(int goalDist) {
  int minF, minS;
  minF = minFront();
  minS = Math.min(minLeft(),minRight());
  int nominal = goalDist / 3; // ILLAH was 2
  if (nominal > 200) nominal = 200;
  if ((minF > 40) && (minS > 20)) {
    return nominal;
  } else if ((minF > 20) && (minS > 10)) {
    return nominal/2;
  } else {
    return nominal/3;
  } }

boolean closeObstacle() {
  return ((minFront() < 17) || (minFLeft() < 8)
          || (minFRight() < 8)); }

int computeRotCloseObstacle(int curRot,
                            int goalRot) {
  if (minLeft() < minRight()) {
    return -50;
  } else {
    return 50;
  }
} // computeRotCloseObstacle() //

// input is between 0 and 3600 and 0 - 3600
// output needs to be -200 to +200 or so...
int computeRotTo(int curRot, int goalRot) {
  int theDiff = goalRot - curRot;
  if (theDiff < -1800) theDiff += 3600;
  if (theDiff > 1800) theDiff -= 3600;
  if ((theDiff > 0) && (minLeft() < 25)) {
   theDiff = 0;
   //System.out.println("left close");
  } else if ((theDiff < 0) &&
             (minRight() < 25)) {
   theDiff = 0;
   //System.out.println("right close");
  }
return (theDiff / 10);
}
```

Figure 10: T.A.W.'s gohome procedure

When `closeObstacle()` was first coded, it was given zero rotational speed and the `else` case directly following it simply set `forwardVel` at a single speed and set `rotationVel` at zero. After testing and debugging, the next step was to add rotation to the `closeObstacle` case and, by inspecting `computeRotCloseObstacle()` you can see that this was performed using one discrete rotation speed for the sake of robot observability.

Next, the else case was changed from an unintelligent "go straight ahead" command to turn toward the goal. Note in `computeRotTo()` that the rotational speed was indeed computed as a continuous function of the difference between desired heading and current heading. While this could diminish robot observability, it affords T.A.W. extremely smooth motion and elicited comments to this effect during the competition.

In spite of this continuously-valued function, T.A.W. achieves a high degree of robot observability by depending on *translational* speed to communicate internal state and perceptual state. When the robot is observed during execution of `gohome()`, it is instantly either moving straight ahead or turning. The observability question is, can an observer tell why the robot is moving the way it is moving?

If T.A.W. is going straight, it is in one of two cases: it believes it is heading in the goal direction, or it cannot turn because it detects an obstacle in `computeRotTo()`. The forward velocity of T.A.W. is significantly faster in the former case (`nominal`) than in the latter case (`nominal/2`), and so this distinction will be obvious.

In the case of turning, T.A.W. turns either toward the goal or away from an obstacle. Turning toward the goal is done at full translational speed (i.e. `nominal`), as shown by `computeForVelTo()`, while turning away from an obstacle is done with a forward velocity of *zero* (based on the `closeObstacle()` case).

The astute reader will note that T.A.W. can speak (`myTongue.playSound("out.au")`). If so, why not program the robot to communicate internal values verbally, just as a C debugger may sprinkle the code with `printf` commands. This is acceptable in many situations, but there are two caveats.

First, there remains a bandwidth limitation on the human observer, and this constrains the number of properties and the number of discrete, chosen actions that are feasible. Second, the robot's entire behavior is often relevant to its goals. In the case of T.A.W., the robot

speaks to the public, so the audio channel is not freely available.

As shown by the `myState` variable, T.A.W. also makes use of a limited amount of state. The property mapping technique was used for each state's behavior, and furthermore the states were disambiguated by designing intentionally different overall robot motion for each state. The speed achieved by T.A.W. during the `GOHOME` state is approximately twice the speed used during the `SERVEFOOD` state.

The control code was designed, implemented and tested over the course of approximately 6 hours of focused work. The result, T.A.W., achieved second place at the 1999 AAAI Competition.

Other, more ambitious mobile robots programmed using this same methodology have demonstrated significant mean time to failure statistics. Chips, in particular, as well as Sweet Lips and Joe, are three full-time tour guide robots that conduct tours in museums. Chips uses a property-mapping, case-based approach to both navigation and obstacle avoidance and has now been running for 22 months autonomously. Its mean time between failure has increased to a stable value of approximately three weeks (Nourbakhsh et al., 1999).

Concluding Remarks

We have used the property mapping technique for more than four years, resulting in several examples of very successful implementations. Our experiences suggest that robot observability is a good predictor of long-term robot reliability. But, is this the most important mechanism by which this technique affects robot reliability?

Property mapping tends to minimize the effective perceptual space via partitioning, and to minimize the effective output range via action sampling. It is conceivable that this form of problem reformulation introduces a real bias toward a space of reliable robot programs, separate from its effect on diagnostic transparency.

The property technique described herein applies to ground percepts and actions. Architectures such at 3T (Bonasso and Kortenkamp, 1996) take advantage of a hierarchical approach in which meta-actions are implemented as primitive behaviors. A next step is to generalize property mapping and robot observability so that they may be applied to such hierarchical systems. This technique also needs to be extended to the multi-threaded case in order to evaluate inherently parallel architectures such as Subsumption (Brooks, 1986).

Finally, we have observed surprising differences in robot reliability when comparing implementations in functional languages (Lisp, ML) and procedural languages (C, Pascal). An examination of whether functional programming languages naturally lead to high degrees of robot observability may be fruitful.

Acknowledgments

Prof. Michael Genesereth teaches CS224 at Stanford University. Profs. Martha Pollack and Don Chiarulli teach CS1699 at University of Pittsburgh. Chris Hardouin created T.A.W. Thanks to Lee Weiss, Iwan Ulrich and the anonymous reviewers for their comments.

References

Bonasso, R. and Kortenkamp, D. 1996. Using a layered control architecture to alleviate planning with inc. info. In *Planning with Incomplete Information*, AAAI Spring Symposium Series.

Borenstein, J. and Koren, Y. 1991. The vector field histogram—fast obstacle avoidance for mobile robots, IEEE J. Robotics and Automation 7 (3) 278-288.

Brooks, R. 1986. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14-23.

Firby, R. 1987. An investigation into reactive planning in complex domains. In *Proc. AAAI-87*.

Halperin, D., Kavraki, L and Latombe J. 1998. Robot algorithms. *CRC Handbook of Algorithms and Theory of Computation*, M. Atallah (ed.), CRC Press.

Horswill, I. 1999. Functional programming of behavior-based systems. In *Proc. IEEE International Symposium on Computational Intelligence in Robotics and Automation*.

Horswill, I. 1993. Polly: A vision-based artificial agent. In *Proc. AAAI-93*, Washington DC, AAAI Press.

Krotkov, E., Simmons, R., Cozman, F. and Koenig, S. 1996. Safeguarded teleoperation for lunar rovers: from human factors to field trials. In *Proc. IEEE Planetary Rover Technology and Systems Workshop*, Minn., MN.

Kunz, C., Willeke, T. and Nourbakhsh, I. 1999. Automatic mapping of dynamic office environments. *Autonomous Robots* 7, 131-142.

Nourbakhsh, I., Bobenage, J., Grange, S., Lutz, R., Meyer, R. and Soto, A. 1999. An affective mobile robot educator with a full-time job. *Artificial Intelligence* 114:95-124.

Nourbakhsh, I., Andre, D., Tomasi, C. and Genesereth, M. 1997. Mobile robot obstacle avoidance via depth from focus. *Robotics and Autonomous Systems* 22, 151-158.

Simmons, R. 1994. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation*, 10:1.