Modeling and Verification of Distributed Control Scheme for Mobile Robots

Mohamed Dekhil, Ganesh Gopalakrishnan, and Thomas C. Henderson

UUSC-95-004

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

April 6, 1995

Abstract

In this report we present a sensor-based distributed control scheme for mobile robots. This scheme combines centralized and decentralized control strategies. Each group of sensors is considered to be a process that performs sensing and carries out local control tasks as well. Besides these processes, there is a central controller that carries out global goals utilizing sensor readings on a need to know bases. In this scheme, the sensors communicate with the central controller, and also may communicate with each other. Communication protocols has been defined and an abstract model for the proposed control scheme was built. Formal verification techniques were used to verify the correctness of these protocols as well as some desired properties of the proposed scheme. The advantages of this scheme over the centralized scheme is that it increases system modularity and flexibility and provides graceful degradation in case of failure of some of the sensors. The results of verifying and simulating this model are presented with a brief discussion and conclusion on these results.

Contents

1 Introduction **2**

1.1 Sensor-based Control 2

1.2 System Verification Techniques 4

2 The Proposed Control Scheme **6**

2.1 Abstract Sensor Model 6

2.2 A Distributed Control Architecture 7

2.3 Communication Protocols 9

3 Experiment and Simulation Results **9**

3.1 Modeling the System 12

3.2 Commanding Sensors and Reaction Control 14

3.3 The Priority Scheme 16

3.4 Implementation 16

3.5 Simulation Results 19

4 Verifying the System **21**

5 Conclusion and Future Work **22**

6 APPENDIX A **23**

1 Introduction

This report presents the steps for defining and constructing a model for distributed sensor-based control scheme, and formally verifying several aspects about the system. The following subsections describe the motivation of this work, and a brief background and the related work in the areas of sensor-based control and system verification techniques.

1.1 Sensor-based Control

In any closed-loop control system, sensors are used to provide the feedback information that represents the current status of the system and the environmental uncertainties. The main component in such systems is the transformation of sensor outputs to the decision space, then the computation of the error signals and the joint-level commands (see Figure 1). For example, the sensor readings might be the current tool position, the error signal the difference between the desired and current position at this moment, and finally, the joint-level command will be the required actuator torque/force.

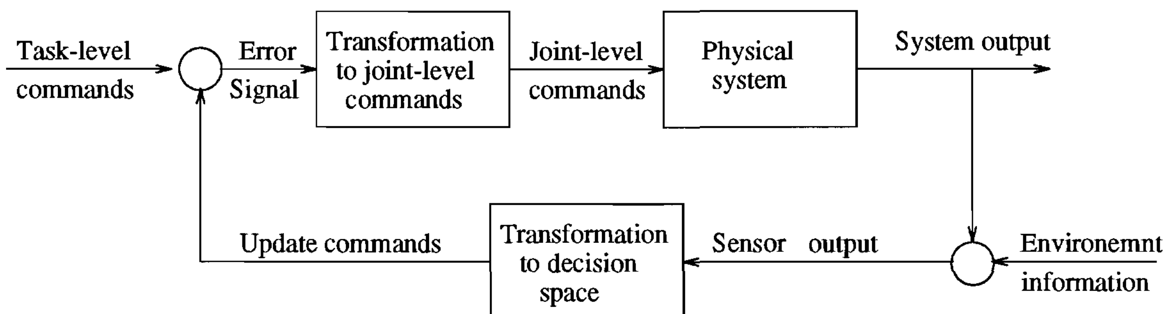

Figure 1: Closed loop control system.

The sensors used in the control scheme shown in Figure 1 are considered to be passive elements that provide raw data to a central controller. The central controller computes the next command based on the required task and the sensor readings. The disadvantage of this scheme is that the central controller may become a bottleneck when the number of sensors increases which may lead to longer response time. By response time we mean the time between two consecutive commands. In some applications the required response time may vary according to the required task and the environment status. For example, in autonomous mobile robot with the task of reaching a destination position while avoiding unknown obstacles, the time to reach to the required position may not be important, however, the response time for avoiding obstacles is critical and requires fast response.

Fast response can be achieved by allowing sensors to send commands directly to the physical system when quick attention is required. This is analogous to human reactions to some

events. In the normal cases, the sensory systems in humans (e.g., eye, ear, nerves, etc.) sends perceived data to the brain (the central controller) which analyze this data and decides the next action to be taken based on the result of the analysis and the required task to be done. However, humans have a very fast contracting reaction when touching hot surfaces for example. In such cases, this reaction behavior is due to commands sent directly from the nerves at the skin spot where the touch occured to the muscles, bypassing the brain.

There has been a tremendous amount of research in the area of sensor-based control including sensor modeling, multisensor integration, and distributed control schemes for robotic applications in general and mobile robots in particular.

A sensor-based control using a general learning algorithm was suggested by Miller [17]. This approach uses a learning controller that learns to reproduce the relationship between the sensor outputs and the system command variables. A technique for sensor-based obstruction avoidance for mobile robots was proposed by Ahluwalia and Hsu [1]. In their technique, the robot is able to move through an unknown environment while avoiding obstacles. Simulations were carried out assuming the robot had eight tactile sensors and the world is modeled as a two-dimensional occupancy matrix with 0's representing empty cells and 1's representing occupied cells. Another method for sensor-based obstruction avoidance was proposed by Gourley and Trivedi [5] using a quick and efficient algorithm for obstacle avoidance.

Hagar proposed a novel approach for sensor-based decision making system [6]. His approach is based on formulating and solving large systems of parametric constraints. These constraints describe both the sensor data model and the criteria for correct decisions about the data.

Lin and Tummala [15] described an adaptive sensor integration mechanism for mobile robot navigation. They divided the navigation process into three phases:

Sensing: firing different sensors then sending the perceived data to the data processor.

Integration: interpreting sensory data of different types into a uniform representation.

Decision: Deciding the action plan based on the current workspace representation.

A distributed decentralized control scheme is proposed by Mutambara and Durrant-Whyte [18]. This scheme provides flexible, modular and scalable robot control network. This scheme uses a non-fully connected control components, which reduces the number of interconnections and thus reducing the number of required communication channels. There has been a fair amount of research in developing languages for sensor-based control for robot manipulators. The goal of such languages is to provide an easy tool for writing adaptive robotic controller. Some of these languages are described in [22]. Several research activities for sensor-based control for robotic applications can be found in [13].

The idea of *smart sensing* was investigated by several researchers. Yakovleff et al. [25] represented a dual purpose interpretation for sensory information; one for collision avoidance (reactive control), and the other for path planning (navigation). The selection between the two interpretation is dynamic depending on the positions and velocities of the objects in the environment. Budenske and Gini [3] addressed the problem of navigating a robot through an unknown environment, and the need for multiple algorithms and multiple sensing strategies for different situations.

Discrete Event Systems (DES) is used as a platform for modeling the robot behaviors and tasks, and to represent the possible events and the actions to be taken for each event. A framework for modeling robotic behaviors and tasks using DES formalism was proposed by Košecká et al. [11]. In this framework, there are two kinds of scenarios. In the first one, reactive behaviors directly connects observations (sensor readings) with actions. In the second, observations are implicitly connected with actions through an observer.

In our proposed control scheme, the sensory system can be viewed as passive or *dumb* element which provides raw data. It can be viewed as an *intelligent* element which returns some "analyzed" information. Finally it can be vised as a *commanding* element which sends commands to the physical system. Each of these views is used in different situations and for different tasks. A detailed description of the proposed control scheme is presented in Section 2.

1.2 System Verification Techniques

In designing and building systems that have several components communicating with each other, verifying their communication protocols arises as an essential step in the design process. Simulation is a widely used method to verify the correctness of a system. The problem with simulation is that it doesn't cover all possible cases that might arise in the real world, which means that simulation can only tell if the system has errors, but it can not determine if the system is error-free.

Formal verification is another approach used to verify the correctness of a system. It requires formulating the system in a mathematical model with some level of abstraction. It also requires a method for specifying system properties to be verified. The problem with this approach is that it is sometimes very difficult to model the system in a precise way, and some of the system properties might not be included in the abstract model. Finite state machines (FSM) and Petri Nets (PN) are the most widely used frameworks to model the system behavior.

In sensory systems with multiple sensors, there must be some protocols to specify the way of communication between these sensors and the controller(s) which uses the sensor readings as a feedback component for the underlying control scheme. Some systems may require coordination among the sensors, which imply that more communication protocols has to be defined. Simulation, and/or formal verification techniques can be used to verify these protocols, and to

check some system properties.

The problem becomes more difficult in concurrent sensory systems when each group of sensors represents a process that can run simultaneously with other processes. Verifying such systems, and concurrent system in general, has been a research topic for several years, and there have been great amount of research efforts in this area.

Automated verification of the logical consistency of communication protocols has been hot research issue for more than two decades. different methodologies has been suggested, and several tools has been developed to assist in the verification process.

Holzmann [8] presented several algorithms for automated protocol verification, and made a comparison between these algorithms based on CPU time requirements, memory usage, and the quality of the search for errors. The four basic types of algorithms he presented are:

1. Exhaustive search.
2. Partial search.
3. Stack search.
4. Memory-less search.

He proposed a new algorithm called *super trace*, based on partial search which works in fixed-size memory and is about two order of magnitude faster than the other methods. He also proposed a framework that combines simulation with formal verification [9, 10]. He developed a tool called SPIN, and a specification language called "PROMELA" which can be used to design reliable protocols by modeling the system and to simulating its behavior, and also by specifying some properties of the system to be verified.

Another framework for automatic system verification was proposed by McMillan [16] who developed another language for this purpose called "Symbolic Model Verification" (SMV). The main contribution of McMillan's work is the possibility of checking large systems with millions of states in a reasonable amount of time and space. In this framework, *temporal logic* is used as the basis for reasoning about concurrent and reactive programs. Temporal logic uses the usual operators of prepositional logic plus other *tense operators* which are used to reason about conditions changing across time. More about temporal logic and its applications can be found in [20, 21]. McMillan used Binary Decision Diagrams [14] to efficiently represent a class of systems, and this approach simplifies the verification process and allows verifying large class of complex problems. Naik and Sistla [19] used the SMV tool to model and verify the Ethernet protocol.

COSPAN is a software system written by Kurshan [12] to assist in the development of large, control oriented programs. Formal verifications is one of the aspects of this language Bradakis [2] proposed a framework for rapid design and verification of reactive behaviors for

sensor-based autonomous agents. In this framework, the behavior of a reactive system is described by a collection of finite state machines which can be implemented to produce real behavior. COSPAN was used to implement the reactive behavior and to formally verify some properties of the system.

An example of a distributed control system of track vehicles was modeled and verified using the Process Algebra techniques by Fischer [4]. In this example, every vehicle and every track section has its own control process. These processes are to communicate to ensure collision free movement of vehicles. This example illustrates the use of automatic formal verification tools in modeling real problems.

2 The Proposed Control Scheme

The robot behavior can be described as a function $\mathcal{F}$ that maps a set of events $\mathcal{E}$ to a set of actions $\mathcal{A}$. This can be expressed as:

$$\mathcal{F}\colon \mathcal{E} \longrightarrow \mathcal{A}$$

The task of the robot controller is to realize this behavior. In general we can define the controller as a set of pairs:

$$\{(e_1, a_1), (e_2, a_2), \ldots, (e_n, a_n)\}$$

where $e_i \in \mathcal{E}$, and $a_i \in \mathcal{A}$

The events can be defined as the interpretation of the raw data perceived by the sensors. Let's define the function $\mathcal{T}$ which maps raw data $\mathcal{R}$ to events $\mathcal{E}$:

$$\mathcal{T}\colon \mathcal{R} \longrightarrow \mathcal{E}$$

2.1 Abstract Sensor Model

We can view a sensory system with three different levels of abstractions (see Figure 2.)

1. **Dump sensor:** which returns raw data without any interpretation. For example, a range sensor might return a real number representing the distance to an object in inches.

2. **Intelligent sensor:** which interprets the raw data into an event using the function $\mathcal{T}$. For example, the sensor might return something like "will hit an object".

3. **Controlling sensor:** which can issue commands based on the received events. for example, the sensor may issue the command "stop" or "turn left" when it finds an obstacle ahead. In this case, the functions $\mathcal{F}$ and $\mathcal{T}$ should be included in the abstract model of the sensor.

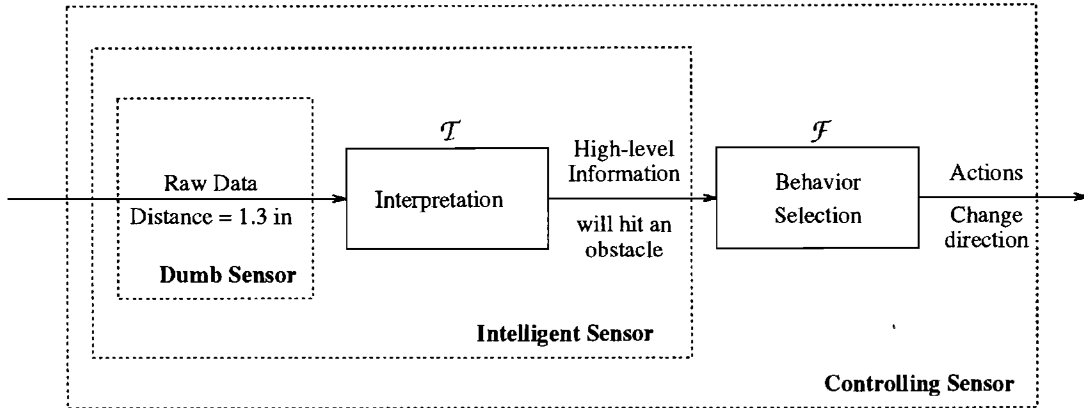

Figure 2: Three levels to view a sensor module.

The dumb sensor can be used as a source for the feedback information required by the control system. It can be also used to gather measurements to construct a map for the surrounding environment. The process that uses a dumb sensor as a source of information needs to know the type of that sensor, the format of the data the sensor returns, and the location of the sensor, to be able to interpret the perceived data. The intelligent sensor may be used for monitoring activities. The process that uses an intelligent sensor. needs to know only the event domain and maybe the location of the sensor. On the other hand, the commanding sensor is considered to be a "client" process that issues commands to the system.

2.2 A Distributed Control Architecture

Several sensors can be grouped together representing a logical sensor [7, 24]. We will assume that each logical sensor is represented as a client process which sends commands through a chanel to a multiplexer (the server process) which decides the command to be executed first. Besides these logical sensors, we might have other processes (general controllers) that send commands to the server process to carry out some global goals. Figure 3 shows a schematic diagram for the proposed control scheme.

Let's call any process that issues commands to the server a em control station. In this figure, there are three types of control stations:

1. Commanding sensors, that are usually used for reaction control and collision avoidance.
2. General Controllers, that carry out a general goal to be achieved (e.g., navigating from one position to another.)
3. Emergency exits, which bypass the multiplexer in case of emergencies (e.g., emergency stop when hitting an obstacle.)

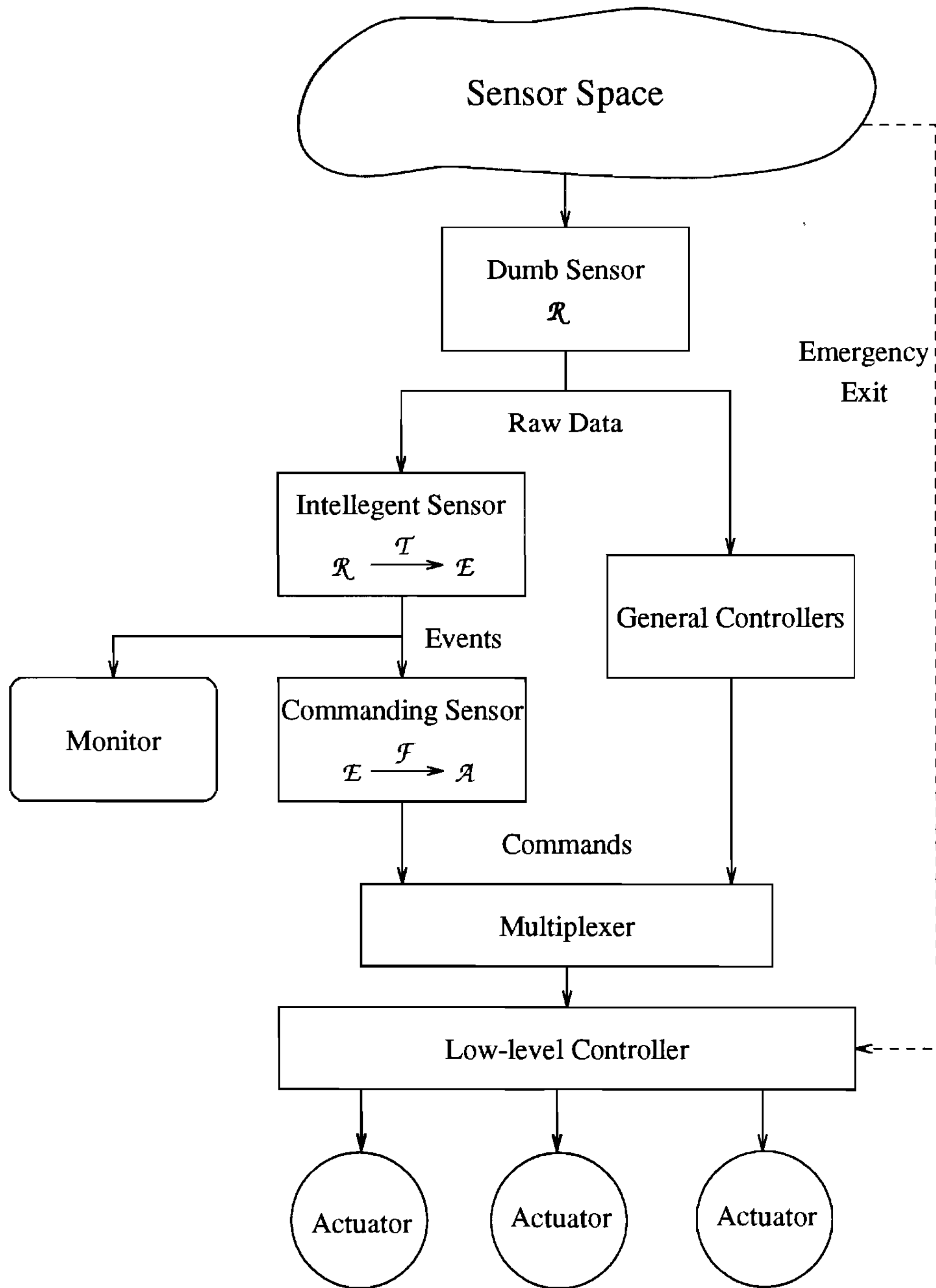

Figure 3: The proposed control scheme.

In most cases, the general controllers require feedback information to update their control parameters. This information is supplied by *dumb sensors* in a form of raw data. On the other hand, a monitoring process might use em intelligent sensors as a source of "high-level" events instead of raw data. All control stations (except for the emergency exists) send the commands to a multiplexer. The multiplexer selects the command to be executed based on a priority scheme which depends on the current state of the system and the type of control station sending the command. Once a command is selected, all other commands can be ignored, since the state of the system will change after executing the selected command.

The low-level controller, shown in Figure 3, translates the high-level commands into low-level instructions which drive the system's actuators. The low-level controller receives its commands either form the multiplexer or from an emergency exit. After the command is executed, the system state is updated, and the sensor space is changed. New sensor readings are received and the cycle keeps iterating.

2.3 Communication Protocols

In the proposed control scheme, there are several control stations that send commands asynchronously to the server. Therefore, we need to define a communication protocol to organize these commands, and to set a priority scheme for selecting the command to be executed first.

Each command station may send commands to the server (the multiplexer) at any time. Each command is associated with the signature of the sender. This signature includes the name and type of the sender, and the priority value. In most cases, the reaction commands (usually from a commanding sensor to avoid collision) has higher priority than a general controller. The priority among commanding sensors may be specified by the user and/or by the current state of the system. Emergency exits should always bypass the server and sends its commands directly to the low-level controller. The priority of the general controllers usually specified by the user.

The global controllers needs to know the current state of the system and the command history to update their control strategy. Therefore, the server has to broadcast the selected command and the current state of the system. The reaction stations might need to communicate with each other to reach a decision.

3 Experiment and Simulation Results

A simulator has been developed to examine the applicability of the proposed control scheme. This simulator is based on a mobile robot called "LABMATE" designed by Transitions Research Corporation (TRC). The LABMATE was used for several experiments at the Department of Computer Science, University of Utah. It also entered the 1994 AAAI Robot Competition [23]. For that purpose, the LABMATE was equipped with 24 sonar sensors, eight in-

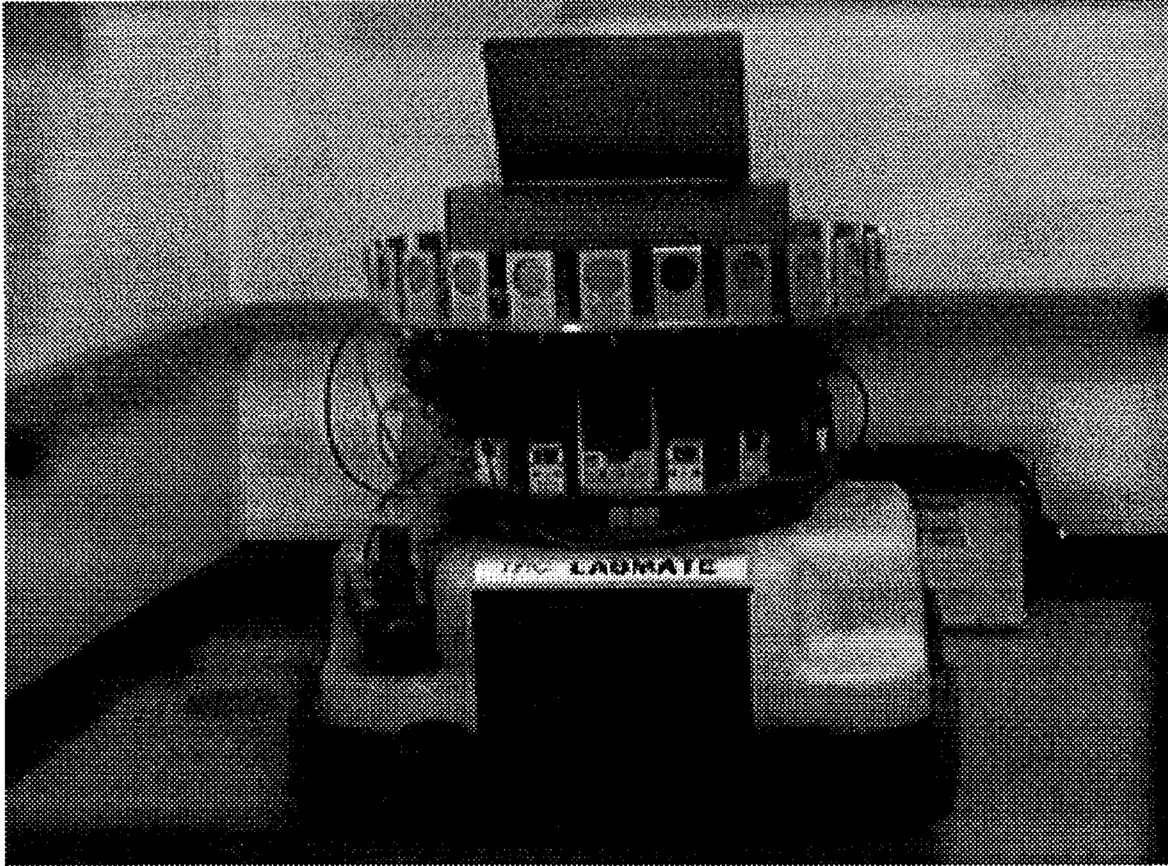

Figure 4: The LABMATE robot with its equipments.

frared sensors, a camera and a speaker.[1] Figure 4 shows the LABMATE with its equipment, and Figure 5 shows a graphical simulator for the LABMATE.[2]

In all previous experiments, the LABMATE was controlled using a conventional control strategy in which there is a central process (the controller) that does everything. This controller receives raw data from the "dumb" sensors, interprets the data, plans for the next move based on these readings and the global goal it has to achieve, and finally issues the required commands. Beside that, the central controller may also produce an output for monitoring purposes. The following are some drawbacks for this scheme:

- The central controller has to know the type and location of each sensor.
- It also needs to know the data format for each sensor type.

[1] The LABMATE preparations, the sensory equipments, and the software and hardware controllers were done by L. Schenkat and L. Veigel at the Department of Computer Science, University of Utah.

[2] This simulator was implemented by A. Efros at the Department of Computer Science, University of Utah.

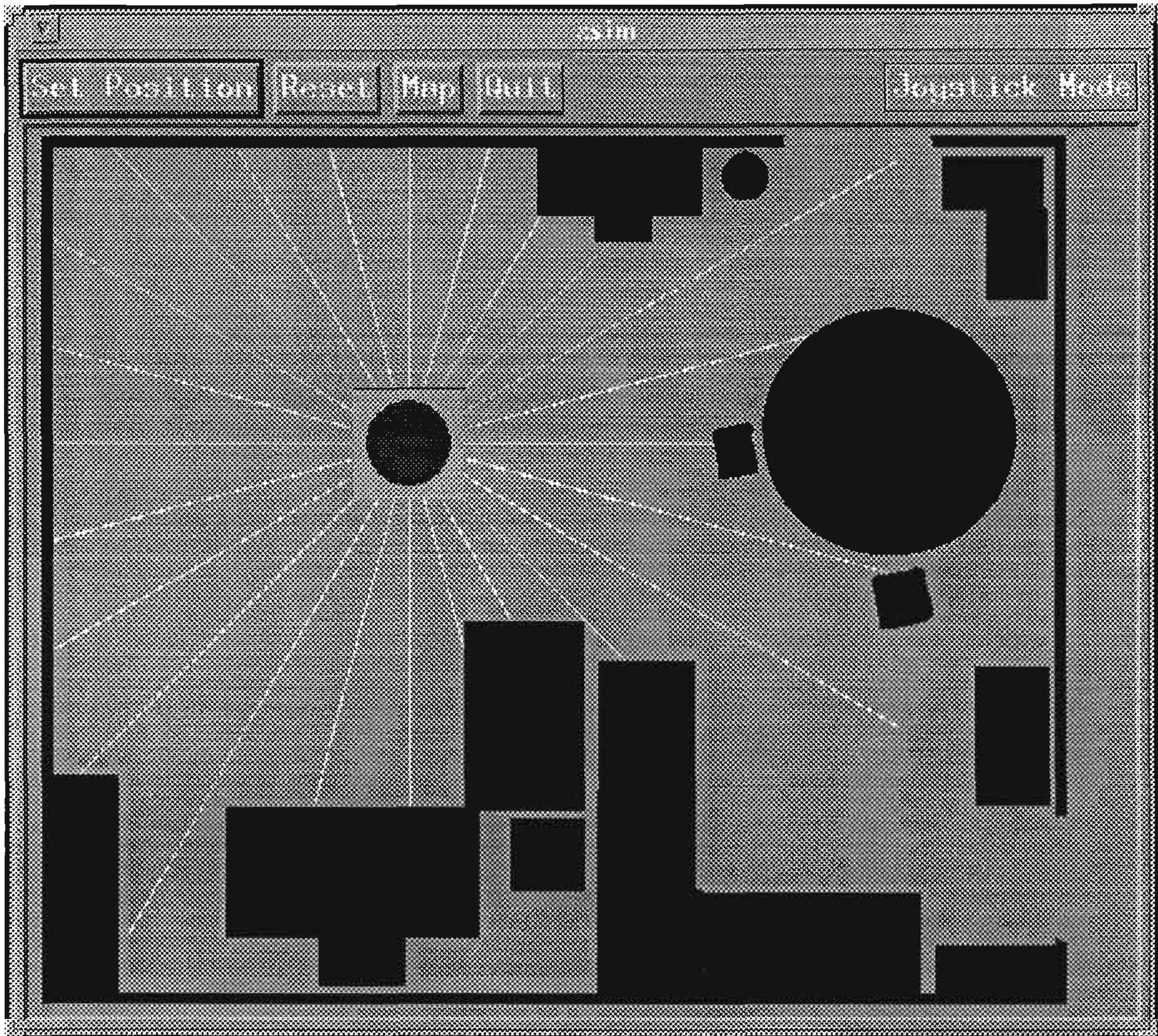

Figure 5: A graphical simulator for the LABMATE.

- It may take long time to issue the required command. This time depends on the interpretation procedure for the data received from each sensor.
- Adding or removing any sensor requires modifying the central controller.

3.1 Modeling the System

The sensors in the old scheme are used only as dumb sensors, while in the proposed scheme, sensors are used in three different levels. They are used as dumb sensors to provide feedback information for a general navigator. They are also used as intelligent sensors providing information to a monitoring process with a speaker as an output device. Finally they are used as commanding stations for collision avoidance. The camera is used as a commanding station which recognize certain objects and guide the robot to collect them. It also provide information to the monitoring process. The emergency exits are hardware bumpers that command the robot to stop if it touch any object. There is also a general controller for navigation and map construction. The commands that can be issued are:

- **GO-FRWD** d**:** move forward distance d inches, where d is a non-negative real number. When $d = 0$, the robot will keep moving forward until other command is issued.
- **GO-BKWD** d**:** move backward distance d inches, where d is a non-negative real number. When $d = 0$, the robot will keep moving backward until other command is issued.
- **TURN-RIGHT** θ**:** turn right θ degrees, where θ is a positive real number.
- **TURN-LEFT** θ**:** turn left θ degrees, where θ is a positive real number.
- **STOP:** stop moving (or turning).
- **RESET:** restart operation after a fault.
- **PICK:** pick up an object.

The system can be in any of the following states:

- **IDLE:** the robot is not moving.
- **FORWARD:** the robot is moving forward.
- **BACKWARD:** the robot is moving backward.
- **RIGHT:** the robot is turning right.

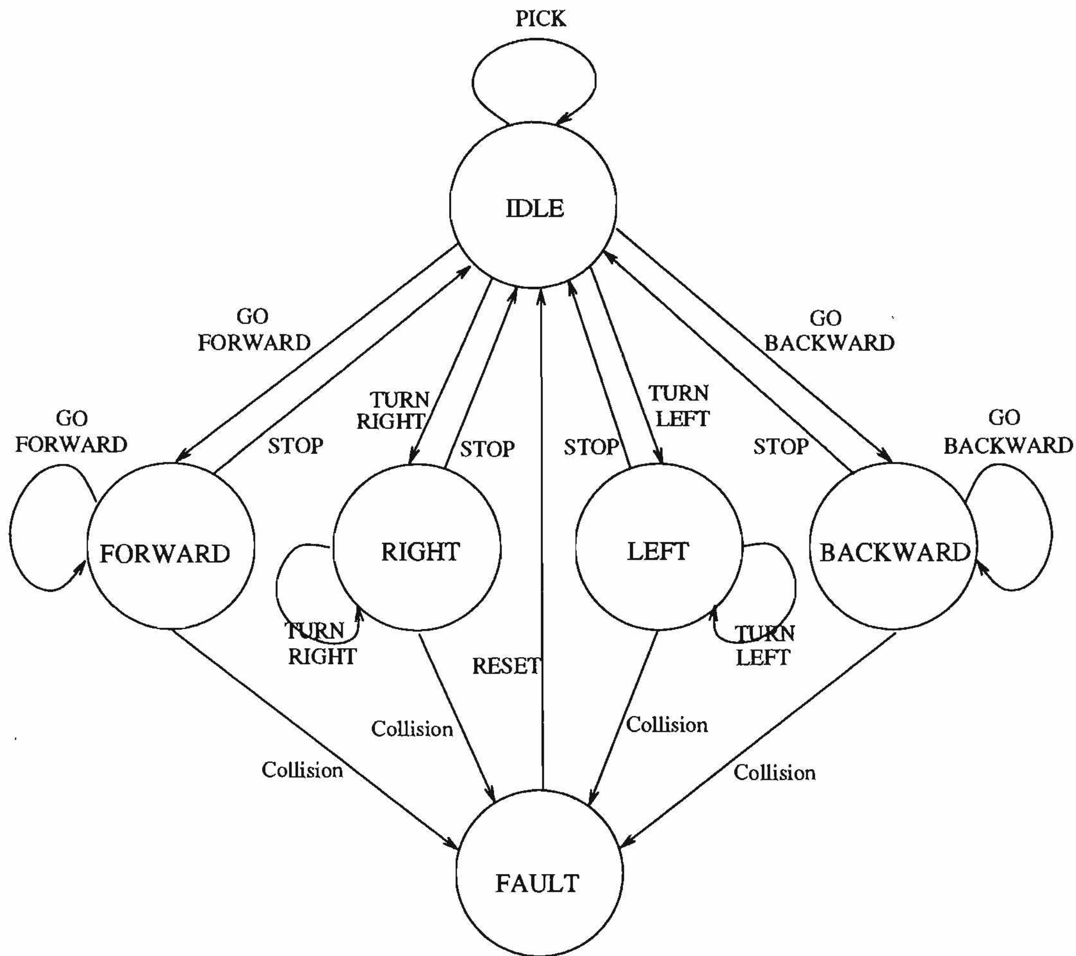

Figure 6: The relation between the system states and the commands.

- **LEFT:** the robot is turning left.
- **FAULT:** the robot hit an obstacle.

Figure 6 shows a state diagram for the system. This figure shows that the robot has to go to the idle state when the command is changed. For example, if the command *GO-FORWARD* is issued, the system will go to the *FORWARD* state and will remain there as long as the following commands are *GO-FORWARD*. Once the next command is different, the system will go to the *IDLE* state first, then it will go to the state corresponding to the current command. This is analogous to what happens in controlling the LABMATE. The LABMATE has to stop first before changing direction. For example, the LABMATE cannot turn left or right while moving forward or backward. In the simulator, this is accomplished by inserting am implicit *STOP* command between any two different commands.

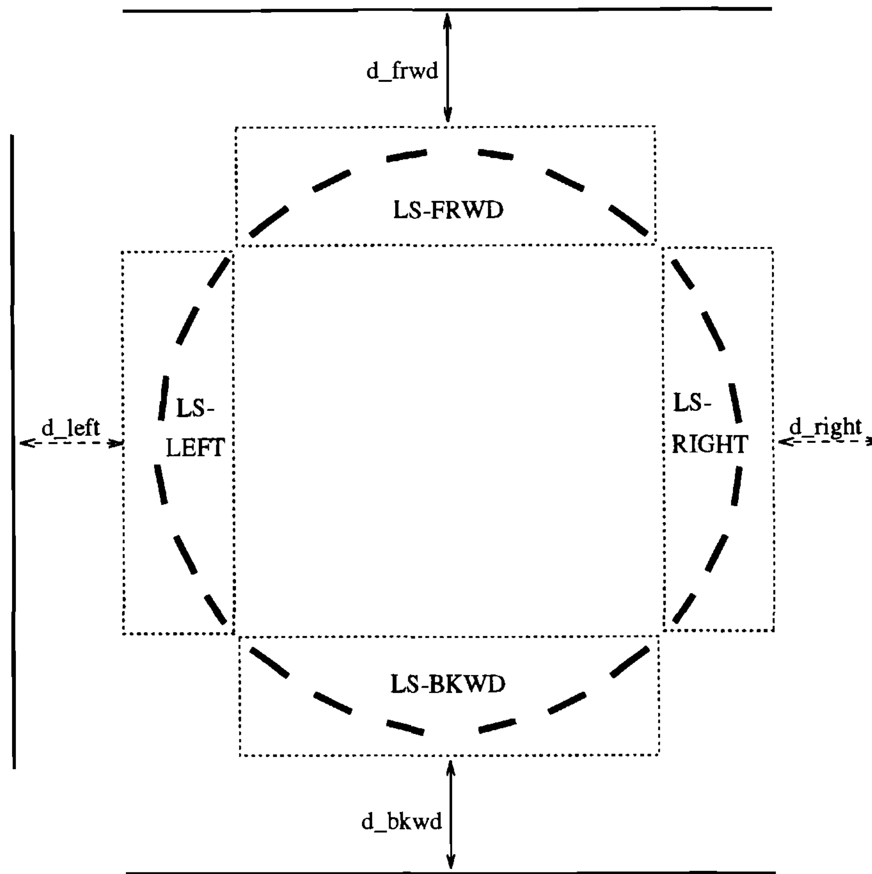

Figure 7: Dividing the sonar sensors into four logical sensors.

3.2 Commanding Sensors and Reaction Control

To simplify our model, the 24 sonar sensors are divided into four logical sensors as shown in Figure 7.

1. **LS-FRWD** consists of the front 6 sensors.
2. **LS-BKWD** consists of the rear 6 sensors.
3. **LS-RIGHT** consists of the right 6 sensors.
4. **LS-LEFT** consists of the left 6 sensors.

These logical sensors communicate with each other to decide the command to be issued. This makes the job of the multiplexer easier, since it will deal with the four logical sensors as one commanding station. The goal of the reactive control is two fold:

1. avoid obstacles.

2. Keep the robot in the middle of hallways, specially when moving through narrow corridors.

We will define two abstract values: *close* (c) and *far* (f). These two values represent the distance between the robot and the closest object at any side. The range for c and f are usually user defined values. The command to be issued as a reaction control depends on the current state of the system and the distance value at each side. There are several ways to define a command function { to achieve the required goal. The assumption here is that there is always enough space for the robot to rotate left of right, therefore we don't need to define any reaction control when the robot is rotating. One such function is shown in Table 1.

d_right	d_left	d_frwd	d_bkwd	*FORWARD*	*BACKWARD*
c	c	c	c	*STOP*	*STOP*
c	c	c	f	*GO-BKWD*	—
c	c	f	c	—	*GO-FRWD*
c	c	f	f	—	—
c	f	c	c	*TURN-RIGHT*	*TURN-LEFT*
c	f	c	f	*TURN-RIGHT*	*TURN-LEFT*
c	f	f	c	*TURN-RIGHT*	*TURN-LEFT*
c	f	f	f	*TURN-RIGHT*	*TURN-LEFT*
f	c	c	c	*TURN-LEFT*	*TURN-RIGHT*
f	c	c	f	*TURN-LEFT*	*TURN-RIGHT*
f	c	f	c	*TURN-LEFT*	*TURN-RIGHT*
f	c	f	f	*TURN-LEFT*	*TURN-RIGHT*
f	f	c	c	*TURN-L/R*	*TURN-L/R*
f	f	c	f	*TURN-L/R*	—
f	f	f	c	—	*TURN-L/R*
f	f	f	f	—	—

Table 1: An example of a decision function for reaction control.

In this table, *TURN-L/R* means the command can be either *TURN-LEFT* or *TURN-RIGHT*, and a dash "—" means no command is issued. Notice that, in case of d_left and d_right have different values, the values for d_frwd and d_bkwd are not important. This is because we need to balance the distance to the left and to the right of the robot, and if, for example, the distance in front (d_frwd) is c, and the robot state is *FORWARD*, then moving to the left (or to the right) will serve both; avoiding the object in front, and balancing the distance on both sides. In the

first case of the table, when the distance is c in all sides, the robot will not be able to move anywhere, and the sensor readings will not change. This will result in a deadlock which requires external help by moving at least one of the obstacles for the robot to be able to move. Figure 8 shows graphically the different cases when the system state is *FORWARD*, and Figure 9 shows the same cases when the system state is *BACKWARD*.

3.3 The Priority Scheme

In this system, there are three commanding stations competing for the server:

1. The four logical sensors representing one commanding station.
2. The camera which guides the robot to pick certain objects.
3. A general controller for navigation.

Beside these commanding stations, there is an emergency exit represented by two bumpers, one on the front and one on the back. As mentioned before, the emergency exits does not compete for the server, rather it sends its commands directly to the low-level controller.

The priority scheme in our application is very simple. The logical sensors have the highest priority, followed by the camera, and finally the general controller. The server checks first for any commands issued by the logical sensors. If there is commands, it will execute them and will keep iterating until no commands from the logical sensors. If no commands form the logical sensors, the server will check for commands issued by the camera commanding station. if there are commands, it will execute one command, then back to check for the logical sensors again. if not, it will check for commands from the general controller, execute one command (if any) then go back for the logical sensors. This routing strategy is shown in Figure 10.

3.4 Implementation

This model was implemented in PROMELA. This language was selected because it is very convenient for simulation and it is easy to add some assertions and verification statements later on after realizing the model.

As a first step in simulating this model, we will consider the four logical sensors as one process that issues commands based on the current state of the system, and the location of the robot with respect to the surrounding objects. In this case, we have four concurrent processes:

1. The multiplexer (the server).
2. The logical sensor commanding station.

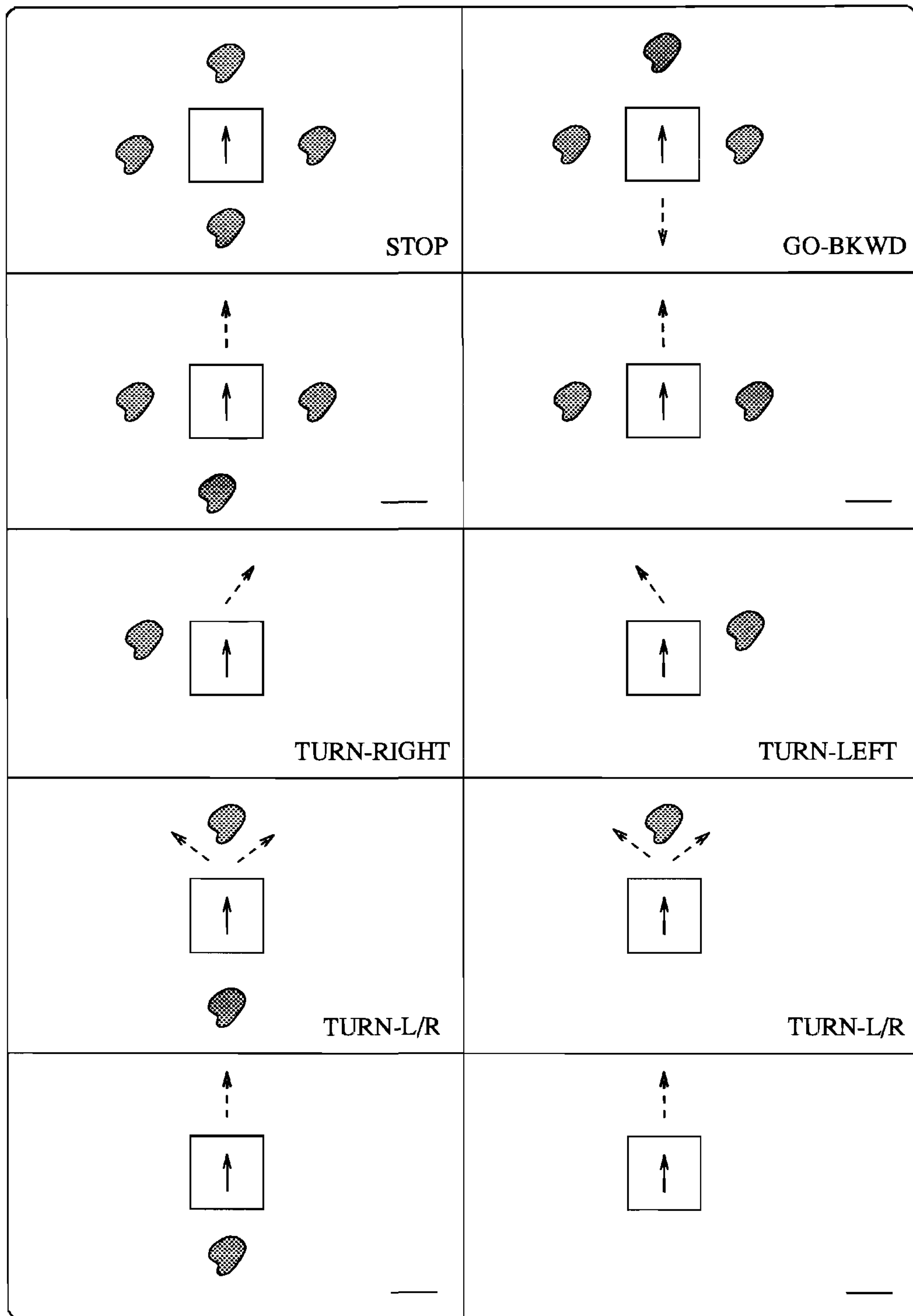

Figure 8: The reaction control when the system state = *FORWARD*.

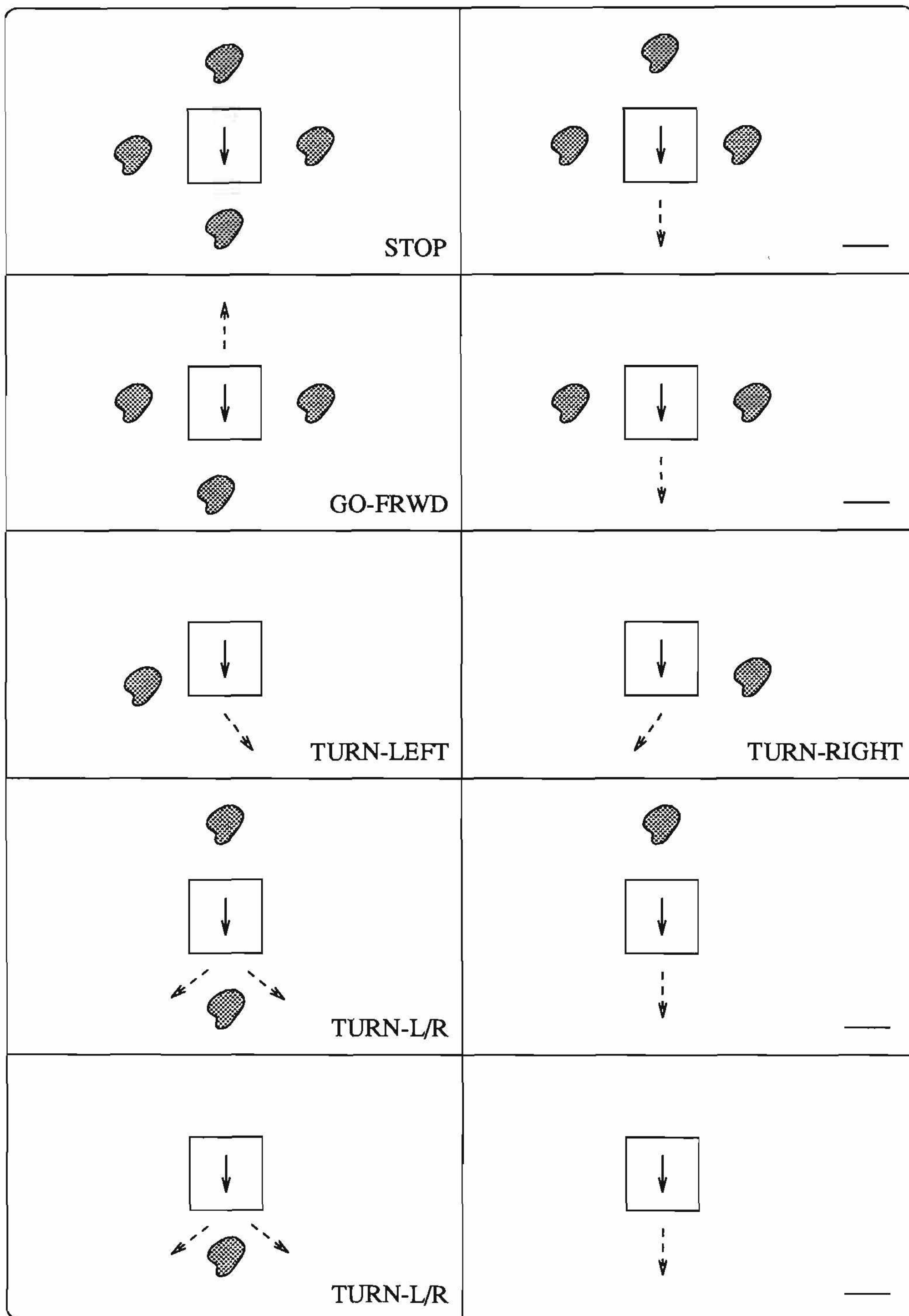

Figure 9: The reaction control when the system state = *BACKWARD*.

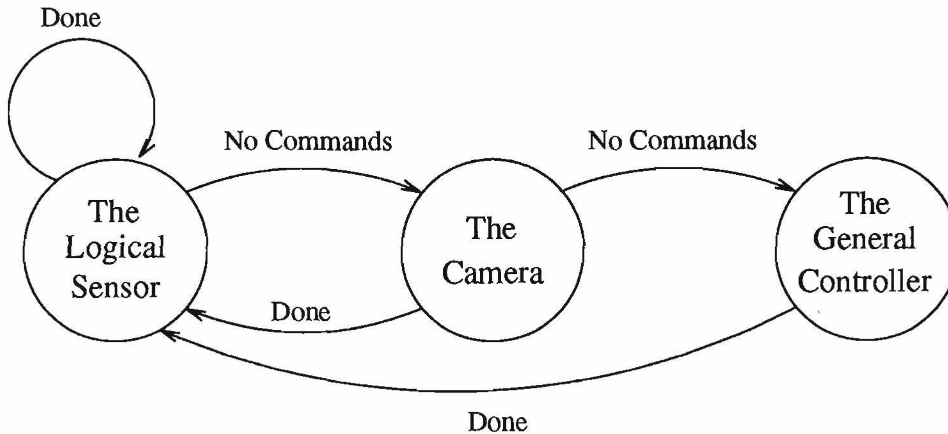

Figure 10: The routing strategy for the server.

3. The camera commanding station.

4. The general controller.

In this simulator, the commands for the general controller are selected at random from {*GO-FRWD, GO-BKWD, TURN-LEFT, TURN-RIGHT*}. the commands for the camera are selected at random from {*GO-FRWD, GO-BKWD, TURN-LEFT, TURN-RIGHT, PICK*}. The values for d_frwd, d_bkwd, d_right, and d_left will be also selected at random, and the decision function in Table 1 will be used to determine the next command. The listing of the PROMELA code for simulating this model is shown in Appendix A.

3.5 Simulation Results

The following is part of the output for the simulator. This output shows which commanding station is currently in charge, and the command it issued.

```
**********************************************************
*                                                        *
*              The LABMATE Simulation program            *
*                                                        *
*              Department of Computer Science            *
*                    University of Utah                  *
*                                                        *
**********************************************************

The System can be in one of the following states:

     0: IDLE
```

```
    1: FORWARD
    2: BACKWARD
    3: RIGHT
    4: LEFT
    5: FAULT

---- The road is clear ... General controller in charge
     The Controller issued the command  GO-BKWD
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-LEFT
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-LEFT
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-RIGHT
@@@@ Object recognized by the camera
     The Camera issued the command  GO-FRWD
**** Avoiding obstacles ... Front=1, Back=1, Right=1, Left=0
     Logical Sensors issued the command  TURN-RIGHT
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-RIGHT
@@@@ Object recognized by the camera
     The Camera issued the command  GO-FRWD
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-LEFT
@@@@ Object recognized by the camera
     The Camera issued the command  GO-BKWD
@@@@ Object recognized by the camera
     The Camera issued the command  PICK
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-RIGHT
@@@@ Object recognized by the camera
     The Camera issued the command  PICK
@@@@ Object recognized by the camera
     The Camera issued the command  GO-BKWD
**** Avoiding obstacles ... Front=1, Back=0, Right=0, Left=1
     Logical Sensors issued the command  TURN-RIGHT
---- The road is clear ... General controller in charge
     The Controller issued the command  GO-FRWD
@@@@ Object recognized by the camera
     The Camera issued the command  TURN-RIGHT
@@@@ Object recognized by the camera
     The Camera issued the command  GO-FRWD
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-RIGHT
---- The road is clear ... General controller in charge
     The Controller issued the command  GO-FRWD
```

```
@@@@ Object recognized by the camera
     The Camera issued the command  GO-FRWD
@@@@ Object recognized by the camera
     The Camera issued the command  TURN-RIGHT
---- The road is clear ... General controller in charge
     The Controller issued the command  GO-BKWD
@@@@ Object recognized by the camera
     The Camera issued the command  TURN-RIGHT
@@@@ Object recognized by the camera
     The Camera issued the command  TURN-LEFT
---- The road is clear ... General controller in charge
     The Controller issued the command  TURN-RIGHT
@@@@ Object recognized by the camera

**** END OF SIMULATION ****
```

4 Verifying the System

The PROMELA language and its compiler "SPIN" provide two modes for running a program: simulation mode, and verification mode. In the previous section, we showed the output of the program under the simulation mode. Now, we need to add some verification statements to the program and run the program under the verification mode.

The following properties of the system will be verified:

- Existence of deadlocks.
- The progress of the server in executing commands.
- Correctness of the commands selected by the logical sensors (See Table 1.)
- Correctness of the server process by checking the selected commanding station.

Checking the progress of the server process as accomplished by adding a "progress" label where the server start executing a command. Checking the correctness of the server in selecting the right commanding station is done by adding the following assertion right after the selection code.

```
assert ((logical_sensor_command != EMPTY &&
         current_command == logical_sensor_command)
      || (camera_command != EMPTY &&
         current_command == camera_command)
      || (controller_command != EMPTY &&
         current_command == controller_command)
      || (current_command == EMPTY))
```

Checking the correctness of the commands issued by the logical sensors is achieved by adding the following assertion at the end of the logical sensors process. Note that this part of code does not check for all the cases, however we can add more conditions to cover all possible cases.

```
assert ((d_left != d_right &&
        (logical_sensor_command == TURN_LEFT ||
         logical_sensor_command == TURN_RIGHT))
     || (d_left == c && d_right == c && d_frwd == c && d_bkwd == c
         && logical_sensor_command == STOP)
     || (d_left == c && d_right == c && d_frwd == f && d_bkwd == f
         && logical_sensor_command == EMPTY)
     || (d_left == f && d_right == f && f_frwd == c && d_bkwd == c
         && (logical_sensor_command == TURN_LEFT
             || logical_sensor_command == TURN_RIGHT))
     || (d_left == f && d_right == f && f_frwd == f && d_bkwd == f
         && logical_sensor_command == EMPTY)
     || (d_left == d_right && d_frwd != d_bkwd))
```

5 Conclusion and Future Work

In this report, a distributed sensor-based control scheme was proposed. In this scheme, each sensor can be viewed with three different levels of abstraction; *dumb sensors* which provide raw data, *intelligent sensors* which provides high level information in a form of events, and finally, *commanding sensors* which can issue commands representing a reaction behavior for the system. Commands can be issued by different processes called *commanding stations*. Each commanding station may issue commands at any time, and a multiplexer (the server) is needed to select the command to be executed. A priority scheme has to be defined as a bases for selection. An example for applying this control scheme to a mobile robot was described along with the simulation results. Finally, several aspects of the model was verified using formal verification tools provided by PROMELA, the language used to implement the model. We believe that this control scheme provides more flexible and robust control systems, and allows more modular design for the whole control system. It also provides fast response for reaction behavior which is an essential requirement in real-time systems.

The next step to this work is to implement a distributed controller for the "real" LABMATE using the proposed control scheme. This requires using Unix machines to drive the robot instead of a PC. A more detailed decision function for the logical sensors may be defined and the communication protocols among the sonar sensors needs to be explicitly defined.

6 APPENDIX A

The PROMELA Code for the Simulator

```
/*********************************************************************/
/* Program: labmate                                                  */
/*                                                                   */
/* Author: Mohamed Dekhil.                                           */
/*                                                                   */
/* Date: December 12, 1994.                                          */
/*                                                                   */
/* Description: This is a simulation for the a sensor-based          */
/*   distributed control scheme for mobile robots. The robot used    */
/*   for this simulation is the LABMATE, a mobile robot made by      */
/*   Transitions Research Corp.                                      */
/*                                                                   */
/*   The program consists of four concurrent processes:              */
/*      (1) The server process.                                      */
/*      (2) The logical sensor process.                              */
/*      (3) The Camera process.                                      */
/*      (4) The general controller process.                          */
/*                                                                   */
/*   Some specs are added to each module to check some of the        */
/*   system's properties.                                            */
/*********************************************************************/

/* ====================== */
/* Defining some constants */
/* ====================== */

#define TRUE 1
#define FALSE 0

#define p 0                           /* Used for the semaphore */
#define v 1                           /* Used for the semaphore */

#define EMPTY -1

#define SIMULATION_LEN 30              /* Length of simulation */

/* Defining the states of the system */

#define IDLE 0
#define FORWARD 1
#define BACKWARD 2
#define RIGHT 3
```

```
#define LEFT 4
#define FAULT 5

/* Defining the commands of the system */

#define STOP 0
#define GO_FRWD 1
#define GO_BKWD 2
#define TURN_RIGHT 3
#define TURN_LEFT 4
#define RESET 5
#define PICK 6

/* Defining the abstract distances "close" c, and "far" f */

#define c 0                         /* Close distance */
#define f 1                         /* Far distance */

/* ============================== */
/* Defining some global variables */
/* ============================== */

int d_frwd; /* The distance from front */
int d_bkwd; /* The distance from back */
int d_right; /* The distance from right */
int d_left; /* The distance from left */

int logical_sensor_command;     /* command from logical sensors */
int camera_command;             /* command from the camera */
int controller_command;         /* command from the general controller */

int current_command;            /* the selected command */

int state;                      /* Current system state */

chan logical_sensor_mutex = [0] of { bit };
chan camera_mutex = [0] of { bit };
chan controller_mutex = [0] of { bit };
chan state_mutex = [0] of { bit };

/* ==================================================================== */
/* Semaphore procedure that mantains several semaphores used in the     */
/* program for mutual execlusion of some variables.                     */
/* ==================================================================== */

proctype Sema4 (chan Ch)
```

```
{
end:
      do
      :: Ch!p
      :: Ch?v
      od
}

/* ================================================================= */
/* The logical sensor module. It sends commands to the server based on    */
/* the current readings and the current state of the system.              */
/* ================================================================= */

proctype logical_sensors()
{

end:
    do
    :: if                    /* Measure the distance at the four sides */
       :: d_frwd = f
       :: d_frwd = c
       :: d_bkwd = f
       :: d_bkwd = c
       :: d_right = f
       :: d_right = c
       :: d_left = f
       :: d_left = c
       fi;

       /* Decide on next command based on the distances and system state */

       logical_sensor_mutex?p;
       state_mutex?p ;

       logical_sensor_command = EMPTY;

       if
       :: (!(state == FORWARD) || (state == BACKWARD)) -> skip
       :: (state == FORWARD) -> skip;
          if
          :: (d_left == c && d_right == c) -> skip;
             if
             :: (d_frwd == c && d_bkwd == c) ->
                logical_sensor_command = STOP
             :: (d_frwd == c && d_bkwd != c) ->
                logical_sensor_command = GO_BKWD
```

```
            :: (d_frwd != c && d_bkwd == c) -> skip
            :: (d_frwd != c && d_bkwd != c) -> skip
            fi
         :: (d_left == c && d_right != c) ->
            logical_sensor_command = TURN_RIGHT
         :: (d_left != c && d_right == c) ->
            logical_sensor_command = TURN_LEFT
         :: (d_left != c && d_right != c) -> skip;
            if
            :: (d_frwd == c && d_bkwd == c) ->
               logical_sensor_command = TURN_RIGHT
            :: (d_frwd == c && d_bkwd != c) ->
               logical_sensor_command = TURN_RIGHT
            :: (d_frwd != c && d_bkwd == c) -> skip
            :: (d_frwd != c && d_bkwd != c) -> skip
            fi
         fi
      :: (state == BACKWARD) -> skip;
         if
         :: (d_left == c && d_right == c) -> skip;
            if
            :: (d_frwd == c && d_bkwd == c) ->
               logical_sensor_command = STOP
            :: (d_frwd == c && d_bkwd != c) -> skip
            :: (d_frwd != c && d_bkwd == c) ->
               logical_sensor_command = GO_FRWD
            :: (d_frwd != c && d_bkwd != c) -> skip
            fi
         :: (d_left == c && d_right != c) ->
            logical_sensor_command = TURN_LEFT
         :: (d_left != c && d_right == c) ->
            logical_sensor_command = TURN_RIGHT
         :: (d_left != c && d_right != c) -> skip;
            if
            :: (d_frwd == c && d_bkwd == c) ->
               logical_sensor_command = TURN_LEFT
            :: (d_frwd == c && d_bkwd != c) -> skip
            :: (d_frwd != c && d_bkwd == c) ->
               logical_sensor_command = TURN_LEFT
            :: (d_frwd != c && d_bkwd != c) -> skip
            fi
          fi
       fi;

       state_mutex!v;
logical_sensor_mutex!v
```

```
    od
}

/* ======================================================================= */
/* The camera module. It sends commands to the server.                     */
/* ======================================================================= */

proctype camera()
{
end:
    do
    :: camera_mutex?p;
       if
       :: camera_command = GO_FRWD
       :: camera_command = GO_BKWD
       :: camera_command = TURN_RIGHT
       :: camera_command = TURN_LEFT
       :: camera_command = PICK
       :: camera_command = EMPTY
       :: camera_command = EMPTY
       :: camera_command = EMPTY
       :: camera_command = EMPTY
       :: camera_command = EMPTY
       fi;
       camera_mutex!v
    od
}

/* ======================================================================= */
/* The controller module. It sends commands to the server.                 */
/* ======================================================================= */

proctype controller()
{
end:
    do
    :: controller_mutex?p;
       if
       :: controller_command = GO_FRWD
       :: controller_command = GO_BKWD
       :: controller_command = TURN_RIGHT
       :: controller_command = TURN_LEFT
       fi;
       controller_mutex!v
    od
```

```
}

/* ====================================================================== */
/* The server module. It checks for commands coming from different       */
/* commanding stations and selects the commands to be executed based on   */
/* a pre-defined priority scheme.                                          */
/* ====================================================================== */

proctype server()
{
  int counter = 0;

     printf ("\nThe System can be in one of the following states:\n\n");
     printf ("     0: IDLE\n     1: FORWARD\n     2: BACKWARD\n");
     printf ("     3: RIGHT\n     4: LEFT\n     5: FAULT\n\n");

     do
     :: (counter < SIMULATION_LEN) ->

        counter = counter + 1;
current_command = EMPTY;

logical_sensor_mutex?p;
        camera_mutex?p;
        controller_mutex?p;
state_mutex?p;

atomic{
if
:: (logical_sensor_command != EMPTY) ->
           current_command = logical_sensor_command ;
           printf(" **** Avoiding obstacles ... Front=%d, Back=%d, Right=%d, Left=%d\n"
           printf("      Logical Sensors issued the command ")
:: (logical_sensor_command == EMPTY) -> skip;
           if
           :: (camera_command != EMPTY) ->
              current_command = camera_command;
              printf(" @@@@ Object recognized by the camera\n");
              printf("      The Camera issued the command ")
           :: (camera_command == EMPTY) -> skip;
              if
              :: (controller_command != EMPTY) ->
                 current_command = controller_command;
                 printf(" ---- The road is clear ... General controller in charge\n");
                 printf("      The Controller issued the command ")
              :: (controller_command == EMPTY) ->
```

```
                printf(" !!!! No Command was issued !! \n")
            fi
        fi
    fi
};
        logical_sensor_command = EMPTY;
        camera_command = EMPTY;
        controller_command = EMPTY;

        /* Execute the selected command */

        if
        :: (current_command == EMPTY)  ->
           printf(" EMPTY \n")
        :: (current_command == GO_FRWD) ->
           state = FORWARD ;
           printf (" GO-FRWD \n")
        :: (current_command == GO_BKWD) ->
           state = BACKWARD ;
           printf (" GO-BKWD \n")
        :: (current_command == TURN_RIGHT) ->
           state = RIGHT ;
           printf (" TURN-RIGHT \n")
        :: (current_command == TURN_LEFT) ->
           state = LEFT ;
           printf (" TURN-LEFT \n")
        :: (current_command == STOP) ->
           state = IDLE ;
           printf (" STOP \n")
        :: (current_command == RESET) ->
           state = IDLE ;
           printf (" RESET \n")
        :: (current_command == PICK) ->
           state = IDLE ;
           printf (" PICK \n")
        fi;

state_mutex!v;
logical_sensor_mutex!v;
        camera_mutex!v;
        controller_mutex!v

     :: (counter >= SIMULATION_LEN) ->
printf ("\n**** END OF SIMULATION ****\n");
break
     od
```

```
}

/* ======================================================================== */
/* The main procedure which will create the required modules.               */
/* and also initializes some global variables.                              */
/* ======================================================================== */

init {

state = IDLE;
logical_sensor_command = EMPTY;
camera_command = EMPTY;
controller_command = EMPTY;

        current_command = EMPTY;

d_frwd = f;
d_bkwd = f;
d_left = f;
d_right = f;

        printf ("\t*******************************************************\n");
        printf ("\t*                                                     *\n");
        printf ("\t*              The LABMATE Simulation program         *\n");
        printf ("\t*                                                     *\n");
        printf ("\t*              Department of Computer Science         *\n");
        printf ("\t*                    University of Utah               *\n");
        printf ("\t*                                                     *\n");
        printf ("\t*******************************************************\n");
        printf ("\n");

atomic {

          /* Start some semaphores */

  run Sema4 (logical_sensor_mutex);
  run Sema4 (camera_mutex);
          run Sema4 (controller_mutex);
  run Sema4 (state_mutex);

          /* Start simulation */


          run server();
          run logical_sensors();
          run camera();
```

```
  run controller()
        }
     }

/* ====================================================================== */
```

References

[1] AHLUWALIA, R. S., AND HSU, E. Y. Sensor-based obstruction avoidance technique for a mobile robot. *Journal of Robotic Systems 1*, 4 (Winter 1984), 331–350.

[2] BRADAKIS, M. J. Reactive behavior design tools. Master's thesis, University of Utah, Mar. 1992.

[3] BUDENSKE, J., AND GINI, M. Why is it difficult for a robot to pass through a doorway using altrasonic sensors? In *IEEE Int. Conf. Robotics and Automation* (May 1994), pp. 3124–3129.

[4] FISCHER, S., AND SCHOLZ, A. Verivication in Process Algebra of the distributed control of track vehicles – a case study. *International Journal of Formal Methods in System Design*, 4 (Feb. 1994), 99–122.

[5] GOURLEY, C., AND TRIVEDI, M. Sensor-based obstacle avoidance and mapping for fast mobile robots. In *IEEE Int. Conf. Robotics and Automation* (1994).

[6] HAGAR, G. D. Task-directed computation of qualitative decisions from sensor data. *IEEE Trans. Robotics and Automation 10*, 4 (August 1994), 415–429.

[7] HENDERSON, T. C., AND SHILCRAT, E. Logical sensor systems. *Journal of Robotic Systems* (Mar. 1984), 169–193.

[8] HOLZMANN, G. Algorithms for automated protocol verification. *AT&T Technical Journal 69*, 1 (Jan. 1990), 32–44.

[9] HOLZMANN, G. *Design and validation of computer protocols*. Prentice Hall, 1991.

[10] HOLZMANN, G. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems 25* (1993), 981–1017.

[11] KOŠECKÁ, J., AND BOGONI, L. Application of discrete event systems for modeling and controlling robotic agents. In *IEEE Int. Conf. Robotics and Automation* (May 1994), pp. 2557–2562.

[12] KURSHAN, R. P. *Discrete Event Systems: Models and Applications*. Springer-Verlag, 1990.

[13] LEE, C. S. G. *Sensor-based robots: algorithms and architecture*. Springer-Verlag, 1991.

[14] LEE, C. Y. representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal 38*, 4 (July 1959), 985–999.

[15] LIN, C. C., AND TUMMALA, R. L. Adaptive sensor integration for mobile robot navigation. In *IEEE International Conference on Multisensor Fusion and Integration* (Oct. 1994).

[16] MCMILLAN, K. L. *Symbolic model checking*. Kluwer Academic Publishers, 1993.

[17] MILLER, W. T. Sensor-based control of robotic manipulators using a general learing algorithm. *IEEE Journal of Robotics and Automation* (Nov. 1987), 157–165.

[18] MUTAMBARA, A. G. O., AND DURRANT-WHYTE, H. F. Modular scalable robot control. In *IEEE International Conference on Multisensor Fusion and Integration* (Oct. 1994).

[19] NAIK, V. G., AND SISTLA, A. P. Modeling and verification of a real life protocol using symbolic model checking. In *Computer Aided Verification, 6th International Conference, CAV'94* (1994), Springer-Verlag, pp. 195–206.

[20] PNUELI, A. The temporal semantics of concurrent programs. In *18th Symposium on Foundations of Computer Science* (1977).

[21] PNUELI, A. Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In *Lecture Notes in Computer Science* (1986), vol. 224, Springer-Verlag, pp. 510–584.

[22] REMBOLD, U., AND HORMANN, K. *Languages for Sensor-Based Control in Robotics*. Springer-Verlag, 1987.

[23] SCHENKAT, L., VEIGEL, L., AND HENDERSON, T. C. Egor: Design, development, implementation – an entry in the 1994 aaai robot competition. Tech. Rep. UUCS-94-034, University of Utah, Dec. 1994.

[24] SHILCRAT, E. D. Logical sensor systems. Master's thesis, University of Utah, August 1984.

[25] YAKOVLEFF, A., NGUYEN, X. T., BOUZERDOUM, A., MOINI, A., BOGNER, R. E., AND ESHRAGHIAN, K. Dual-purpose interpretation of sensory information. In *IEEE Int. Conf. Robotics and Automation* (1994).