Yale University
Department of Computer Science

ARS MAGNA
The Abstract Robot Simulator Manual

Version 1.0

Sean P. Engelson **Niklas Bertani**

YALEU/DCS/TR-928
October 1992

This work was partially supported by the Defense Advanced Research Projects Agency, contract number DAAA15-87-K-0001, administered by the Ballistic Research Laboratory. The first author is supported by a fellowship from the Fannie and John Hertz Foundation

Ars Magna
The Abstract Robot Simulator Manual
Version 1.0

Sean P. Engelson **Niklas Bertani**

Abstract

AI planning research has historically operated in formal abstractions of the real world. This approach was useful in discovering many fundamental issues underlying planning; also, problems in simple domains such as the blocks world can turn out to be surprisingly difficult. Lately attention has turned to planning for more realistic domains in which micro-world simplifying assumptions do not hold. This shift of focus introduces a new problem of validation and comparison of different planning theories and systems. A proper domain for planning problems must be realistically complex but also simple enough to support controlled experimentation.

To address these questions, we developed the Ars Magna robot simulator. The simulator provides an abstract world in which a planner controls a mobile robot. Mobile robotics is a particularly apposite domain since it is a major application area for AI planning techniques. Ars Magna's environment and robot models are based on current robotics research, so that the domain is reasonably realistic. At the same time, we abstracted away from many (though not all) real-world details of kinematics and motor control. Experiments may be controlled by varying global world parameters, such as perceptual noise, as well as building specific environments in order to exercise particular planner features. The world is also extensible to allow new experimental designs that were not thought of originally.

This work was partially supported by the Defense Advanced Research Projects Agency, contract number DAAA15-87-K-0001, administered by the Ballistic Research Laboratory. The first author is supported by a fellowship from the Fannie and John Hertz Foundation

Contents

Chapter 1

Introduction

For many years, classical AI planning research concerned itself with relatively simple formalised abstractions of the real world. This was necessary to be able to get a grasp on the issues underlying planning; the methodology was also sufficient in that problems in seemingly trivial domains, such as the blocks world, turn out to be surprisingly difficult. Lately, however, attention has turned to planning for more realistic domains, such as mobile robotics, in which micro-world simplifying assumptions do not hold. This shift of focus introduces a new problem of validating and comparing different planning theories and systems [2]. The difficulty is in devising a domain for planning problems that is at the same time realistically complex and simple enough to run controlled experiments in.

As an attempt at dealing with these problems, we have developed the ARS MAGNA[1] robot simulator. The simulator provides an abstract world in which a planner may control a mobile robot. This domain is particularly appropriate since there is growing interest in mobile robots in the AI community as a major application area for AI planning techniques. We designed environment and robot models based on current robotics research, so that the domain would be realistic. At the same time, we attempted to abstract away from many (though not all) real-world details of kinematics and motor control. Experiments may be controlled by varying global world parameters, such as perceptual noise, as well as building specific environments in order to exercise particular planner features. The world is also extensible to allow for the development of new experimental designs. While we hope that the simulator will be generally useful, our design was influenced by recent work in reactive planning, particularly Firby's RAP system [4] and McDermott's RPL language [7], as well as Agre's work on deictic representation [1].

[1] Abstract Realistic Simulation of Mobile robots for Analyzing Goal-achievement, Navigation and Adaptation

1.1 Simulated Domains

When designing an environment for experimentation in planning, there are a number of issues of that must be addressed. There are three issues of *versimilitude* which describe the ability of a domain to probe the limits of planning methods. First off, the world must be sufficiently *complex* so that interesting problems may be posed. Second, the world must not be completely *controllable* by the robot; there must be some uncertainty in the outcomes of its actions. And third, the world should not be completely *observable*; questions of sensing and reasoning about knowledge are finessed away in perfect-information domains. These requirements all stem from the desire to evaluate planning systems realistically—the real world is complex, incompletely controllable, and incompletely observable. We chose to model a mobile robot in an indoors environment, basing our model on current research in robot sensing and control. This choice satisfies the criteria above well—the world is both geometrically and ontologically complex, robot actions may fail in realistic ways, and sensors are local and noisy. One ancillary requirement stemming from real-world conditions is that control over time properly lies in the simulator, not in the planner; ideally they should be implemented as separate processes. The ARS MAGNA simulator is implemented using processes in Lisp, providing asynchronous planning and action, with time controlled independent of of the planner.

From the standpoint of the experimenter, the simulator must support *variability*, so that complex experiments may be designed to elucidate particular features of planning systems under consideration. We distinguish two types of variability: *flexibility*, such that a wide variety of different test problems may be posed, and *tunability*, where various system parameters may be changed within a particular test scenario (eg, sensor noise). ARS MAGNA provides flexibility in several ways: world structure can be specified, objects may have user-defined attributes, and manipulation methods may also be defined. This all allows arbitrarily complex scenarios to be constructed, within a uniform simulation framework. ARS MAGNA also provides a set of tunable parameters, controlling sensor and effector noise, action speed, and robot dynamics. These parameters correspond to physically meaningful concepts, and so may be varied meaningfully during experimentation.

1.2 Domain Design Issues

Examining current areas of planning research allows us to identify specific issues that should be addressed in a simulation domain for planning. Several of these have been identified by Al-Badr and Hanks in their critique of the Tileworld domain [2]. The analysis below is abstracted, to some extent, from their discussion. They identify the following current research problems in planning: exogenous events, the time cost of planning, richness of world models, sensing and effecting, measuring plan quality, and multiple agents. Our work currently addresses only the first four of these issues. The question of measuring plan quality is, as Al-Badr and Hanks note, properly the concern of the planning researcher—'goodness' is not built into the world. Support for multiple agents is conceptually simple in the ARS MAGNA framework, and will probably be incorporated sometime in the near future.

Exogenous events

The problem of exogenous events is how an agent deals with events in the world not under its control. Other than the fact that such events occur, we can identify three important qualities that exogenous events may have in a given domain. The first is *complexity*—are the events structurally simple (eg, tiles appearing) or complex (eg, prey hiding from the agent). The second quality is *variability*—how many different sorts of events are possible? It is conceivable to have a domain with only one event; regardless of that event's complexity, great planner flexibility need not be required. Thirdly, we wish events to be *inhomogeneous* and treat different parts of the world differently. An example of a homogeneous event is the periodic 'wind' in the NASA Tileworld [8]. Homogeneous events are easier to deal with, since their effects on different objects are similar. ARS MAGNA supports exogenous events through *kickers*[2], intermittently applicable procedures that may move or change objects in arbitrary ways. Kickers are implemented as Lisp functions, and so inherit the full complexity and variability of Lisp. Direct support is provided for common operations, such as moving objects around. Since kickers are applied (usually) to specific objects, inhomogeneity is assured.

Handling time

There are three issues to be addressed as far as time handling goes: independence of simulator time from thinking time, synchronization between the world and the planner, and the ability to 'skip ahead' and speed up the simulation. To achieve 'temporal independence', we use multiple threads of execution; simulator time proceeds as a sequence of 'ticks', in each of which a quantum of action takes place in the world. Synchronization is achieved by having robot actions call planner-supplied continuations when important events occur. This method is open to abuse, due to its flexibility (arbitrary bits of code can be executed in the simulator process space), but detecting this is easy, so a modicum of discipline suffices to keep things clean. Time skipping is not currently supported in ARS MAGNA, but it is easy to conceive of it being implemented by increasing the action quantum. This would have the disadvantage of reducing the temporal resolution (and hence the accuracy) of the simulation, but that problem is inherent when skipping ahead in a simulation (unless an exact closed-form solution is available, which it is not here).

Model richness

In analysing the richness of a world, we ask the following questions. Are many different sorts of objects possible? Furthermore, are these different objects related to each another in interesting ways (eg, categorization or functional interaction)? And can

[2]Thanks to Drew McDermott for this term.

complex causal mechanisms be devised, either in terms of particular objects changing over time and via interactions with the agent or the world (eg, eggs + mixing + frying → omelette), or in terms of devices built out of other objects (eg, chassis + engine + transmission + wheels → car)? ARS MAGNA, as noted previously, supports an unlimited variety of objects in the world. Conceptual relations between objects are properly in the agent's mind, in terms of the relations between object attributes. So, a bird is an object with wings and feathers (say). Functional relationships can be established by world designer-defined manipulations—for example, the ability to lock a door is a manipulation that relates a key to the door. Kickers can be easily used to define internal causal properties of objects. Composite objects are not currently supported, but could probably be devised with some ingenuity (say, by a manipulation that uses two objects, creates a third, and destroys the first two).

Sensors and Effectors

As we consider the sensors and effectors by which the agent interacts with its world, we must consider four issues: bandwidth, noise, cost, and interface cleanliness. By bandwidth we mean the amount of information the agent can gather or change about the world, compared with the information actually there. For example, a color-blind robot would not fare well in a world in which color was a functionally important cue. Sensors and effectors must also be noisy to be realistic; we should also like the noise level(s) to be adjustable in a meaningful way. Furthermore, sensing and effecting cost the agent, at least in terms of time. Such costs must be part of the domain model, and again, should be adjustable. And finally, for purposes of controlled experimentation, the interface between the agent and the world must be clean, well-defined, and documented. Without this requirement, it is hard to be entirely clear about what an experiment really measures. We have attempted to address all of these concerns in ARS MAGNA. Objects in the world are defined by their location and attributes, all of which can be sensed and modified (through manipulations). In fact, ARS MAGNA's object sensing models a combination of both top-down and bottom-up sensing, allowing complex perceptual interactions to be specified by the designer. Other sensors are provided for uncertain odometry, coarse perceptual matching and for categorizing types of environmental structures. Range sensors are not currently modelled; however, at the level of abstraction that we are operating at, this is not a severe loss. All sensors and effectors have associated noise and error modes, all of which can be adjusted by the experimenter. One of our most important system design goals was that the agent-world interface be clean and conceptually simple. All sensor/effector operations are specified similarly, using a small set of predefined data types when complex information is communicated. User-defined manipulations can be defined in a structured manner, and are treated similar to built-in operations.

1.3 Some Other Domains

This section is not a comprehensive review of the literature. Rather, for context, we discuss some systems with similar goals as ARS MAGNA in terms of the criteria above.

Tileworld

One recent effort whose goals are similar to ours is Pollack and Ringuette's Tileworld simulator [9]. The Tileworld domain is a game played on a grid, where the robot's objective is to push square 'tiles' into polyomino-shaped 'holes'. Holes have limited capacity, requiring reasoning about limited resources. Further, tiles, holes, and obstacles can randomly appear and disappear, providing some uncertainty in the world. Various parameters of the system may be adjusted via 'knobs', allowing for controlled experimentation. The Tileworld is, however, fairly simple and controlled, as well as completely observable (the robot is omniscient), as Al-Badr and Hanks point out in [2], and hence it is hard to say whether it supports addressing fundamental issues of world complexity, incomplete knowledge and prediction, and real-time planning.

Seaworld

Another simulator built as a testbed for AI system experimentation is Vere and Bickmore's Seaworld [12]. The simulator was developed for use in testing their integrated AI agent, whose goal was to explore issues of integrating techniques for planning, natural language comprehension/generation, knowledge representation, and episodic memory. The agent controls a robotic submarine which receives natural language commands over a simulated 'radio link'. The environment is reasonably complex, with a wide range of different objects with which the agent can interact. Further, full 2D geometry is used; the robot does not live in a discretized space. However, this complexity is not easy to control, thus making it difficult to perform controlled experiments for comparative analysis. Experimentation appears to consist primarily of 'anecdote analysis'—seeing if what the agent does in a few particular scenarios makes sense. Also, like the Tileworld, action and perception are error-free, finessing a crucial issue in real-world systems.

Truckworld

The Truckworld domain developed by Firby and Hanks [5] satisfies most of our criteria reasonably well. The simulator provides for exogenous events via update methods that can change objects after each robot action. Time is handled by skipping the simulator ahead by a number of time steps after each action; unlike the Tileworld, however, the duration of each action is determined by the simulator, not the planner. Simulated environments are graphs of places connected by roads, which further points up the atomic action model. Other than generic visual scanning (Is there an object in my vicinity?), no true action parallelism exists. The Truckworld simulator is implemented using an object-oriented methodology, which allows for unlimited model richness, by allowing the user to write their own methods for object manipulation and evolution. Agent resources are also integrated into the world—movement uses up fuel. Sensing and effecting are error-prone with adjustable error-rates.

The Truckworld approach to time has some difficulties, both of implementation (update methods must reason about different lengths of time explicitly) and of operation, in that actions are essentially atomic (nothing happens during them), though they may take some time. The main difficulties with the Truckworld are in using it for controlled experimentation. There is little structure for defining and customizing objects in the world, due to the general nature of the object-oriented methodology. This makes it difficult to establish systematically qualitatively vary an experimental setup. Effector error is limited to arm clumsiness, speed variability, and movement failure; it is difficult to vary these parameters in a systematic fashion. Furthermore, it is hard to relate these to generic physically meaningful concepts. The generality of the object system makes it difficult to systematically vary perceptual error, since errors are hidden inside object-specific sensing methods.

1.4 A Change of Focus

All of the questions discussed in this section, many of which Ars Magna addresses, can be summarized as a desire for a change of focus in planning domain testbeds. In the early days of AI, game-like domains were seen as desirable objects of study, due to their simplicity [11]. Recently, though, as many fundamental issues of planning have become better understood, this same domain simplicity has become a liability, in that the new frontier is dealing with complexity itself. A number of specific types of complexity have been discussed above; undoubtedly, new areas of inquiry will open up in the future. Be that as it may, this conceptual shift demands a similar shift in research agenda, away from game-like domains towards life-like domains. Game-like domains are characterized by determinism, enumerable rules, well-defined goals, and bounded extent. By contrast, the life-like domains that must begin to be addressed are non-deterministic, have uncertain rules of operation, agent-defined, often vague, goals, and unbounded extent. These are the issues that the next generation of intelligent systems must deal with, and development of appropriate life-like problem domains for experimentation are essential.

Chapter 2

Basic Concepts

As discussed in the introduction, one of the main motivations behind the design of the ARS MAGNA simulator is to provide a realistically complex, though tunable, environment for planning research. We therefore chose to model a mobile robot moving about in an indoors environment[1]. The robot itself is modelled as a point robot with sensors, effectors, and internal carrying capacity. These can be customized to experiment with different robot configurations. The robot moves about in a world containing *walls* and *things.* Walls are unmovable obstacles and give environmental structure, through the notion of *place types.* A place type denotes a class of world positions with a particular geometric obstacle structure, eg, doorways. Things represent objects in the world and have a set of attributes, some of which can be sensed, and some of which affect the robots actions or movement. The remainder of this section discusses these basic components in more detail.

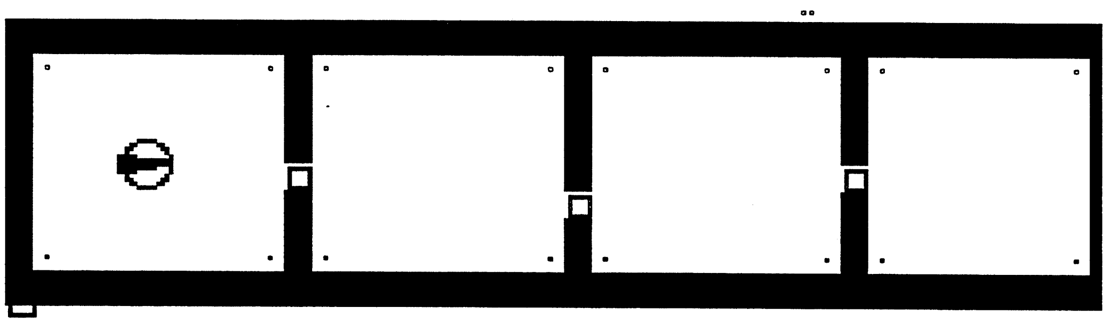


Figure 2.1: A simple ARS MAGNA world.

[1]The world model can also be tailored, and can probably approximate other robotic environments as well; we have not explored this possibility yet, however.

2.1 World

The robot lives in a planar world; a simple example of such a world is in Figure 2.1. One simplification that has been made is that the world is tesselated into a grid. By convention, we call the length of the side of a cell a meter. All entities in the world do have real-valued positions; the grid is only used for limited purposes. First, grid cells may be either filled or empty, filled cells denoting walls. Second, an empty cell may be have an associated place type, depending on the surrounding wall structure. The map in Figure 2.1 has examples of cells of two place types: doorways (marked by squares) and concave corners (marked by dots). Place types can be sensed by the robot; this is described further below. Place types also have *canonical directions* that can be used for stabilizing the robot's configuration. The grid is also used internally for indexing objects and the robot. One notion this helps with is *locality*—an object is local to the robot if it is less than one meter away; the grid makes such objects easy to find.

In the real world, many sensors exist which can sense various properties of physical objects. We abstract away from this complexity and assign to every wall cell and object a single *substance*, consisting of a symbolic *type* and numeric *parameter*. This is intended to model some sort of high-level sensory processing that categorizes substances (eg, brick-like) and measures some property of it (eg, texturedness). In the future, this may be extended to more complex perceptual descriptions. In addition to a substance, objects (as opposed to walls) have a set of attributes which describe the object. Such attributes include, for example, size, weight, and whether or not the object obstructs vision or movement. These attributes can also be sensed by the robot, as described below. The world is made dynamic by the fact that objects can move about by themselves; each object may have associated with it a set of *kickers* which occasionally move the object about.

2.2 Robot

The simulated robot is a point robot with essentially synchrodrive kinematics. It can turn in place at a limited angular velocity, and can move forward in its current direction. There are no explicit bump sensors, but movement into an obstacle signals an error. There are two main kinds of sensing. First, the robot can 'directly' sense the place type of its current cell, if any (with noise). Second, the robot can do more detailed and long-range sensing using *cameras*.

Each camera has a limited field of view and a particular resolution. One form of camera sensing is acquisition of *designators* of objects or places. A designator encapsulates percept-relative information the robot has about a particular entity in the world, enabling the robot to approach the entity, manipulate it, and so forth. A designator for a place can be acquired given a specification of the place's type (eg, 'doorway'), given that such a place is in the camera's field of view. The robot can acquire a designator on an object by specifying a partial description of the attributes of the object desired, which description is matched (with noise) to the actual attributes of objects in the field of view; one matching object is chosen to be 'seen'. False designators, denoting nothing real (ie, hallucinations), can also occur. Designators on objects come in two flavors, local and non-local. Local designators only apply to objects next to the robot; such designators can be used for manipulating the objects they denote. Non-local designators, on the other hand, can only be used to allow the robot to approach the objects they denote. Note that cameras are used to track designators—when objects move, tracking cameras follow; when they are obscured (and sometimes randomly) designators are lost. The other form of camera sensing is the *view*. A view is a list of numbers, samples of the substance parameters of the visible (occluding) surfaces in the camera's field of view. The view is thus an abstraction of the notion of image signature developed in [3] for place recognition. Views can be compared and matched for place recognition.

Object manipulation is done using one of a set of hands (really arms with grippers), each of which has a limited strength. A hand can grasp an object given a local designator, if the object isn't too large. Hands can also be inserted into containers to get at things inside. Objects can be transported by carrying them in a hand, but we also provide storage bins on the robot into which objects can be placed. Bins can also be closed to prevent objects from kicking themselves out.

2.3 Time

Time is discretized into a sequence of *ticks*, each corresponding to a quantum of action. Time in the simulator runs asynchronously to 'thinking' time; calls to robot commands return immediately, after queuing events in the simulator's time stream.

Chapter 3

Robot Programming

3.1 Preliminaries

ARS MAGNA is written in Nisp [6], a portable macro package with compile-time typing. Nisp code looks mostly like regular Common Lisp code, with some exceptions[1]. As far as this manual is concerned, the relevant differences are:

- Variables may be declared when bound. A type is declared by putting a hyphen followed by a type designator after the variable's name. Type designators are either type names (in upper-case here, by convention) or complex type descriptions, such as `(Lst` *t*`)`, which refers to the type 'lists of objects of type *t*'. `VOID` is the type denoting 'no values'. An object of type `(~` *t*`)` is either `#F` or of type *t*.
- If *x* is if type *t*, `(!_`*slot x*`)`, `(!_`*(t slot) x*`)`, or `!>`*x.slot* accesses the given slot of *x*.
- Nisp procedures are defined using `defproc` and `deffunc` (with and without side-effects, respectively). The procedure's result type is shown after the procedures name, preceded by a hyphen.
- An object of type *t* is made by `(make` *t -args-*`)`.
- The macro `!=` is essentially a synonym for `setf`.
- `(\\` *-args- -body-*`)` is the same as `#'(lambda` *-args- -body-*`)`, except that NISP type declarations are allowed in *-args-*.
- Boolean values are given as `#T` and `#F`, for clarity.

All simulator symbols are external in the `ARS` package, except where otherwise noted.

3.1.1 Continuation-passing

Since the simulator's execution is asynchronously parallel to the robot's program, some mechanism must be used to return values and signal errors. In particular, a robot

[1]This discussion is modified from that in [7].

command (say, "move forward 2 meters") may cause an error (say, "path obstructed") long after the command is issued and the robot program has moved on. The way we have chosen to address this issue is to use continuation-passing for asynchronous 'communication' between the separate execution streams. This works as follows. Each robot command is passed, in addition to its usual arguments, a *continuation*, a function of one or more arguments which will be called when the command terminates for any reason. This function thus specifies how to 'continue' computation. The first argument to a continuation is an error code which specifies why the command terminated (`NIL` indicates no error and successful termination). There are a number of different error codes for different situations, they are described below. Other continuation arguments give command-specific parameters which may be useful. For typing purposes, instead of creating continuation closures with `lambda` or `\\`, please use `continuation` in exactly the same way.

3.1.2 Resources

Another issue is the fact that the robot can't do two things at once. Actually, this isn't quite true, because cameras can operate independently of the hands and each other and so on. However, since a robot command may be issued before a previous one has had a chance to complete, conflicts may occur. This is prevented in the ARS MAGNA system by associating commands with particular *resources*, such that only one command may be active on a resource at a time. Resources are simply the robot 'parts' involved in the action—for movement, the resource is the robot itself, for perception, the particular camera being used, and so on. By convention, the resource for a command is its function's first argument. When a new command is issued on a resource before a previous command had a chance to complete, the previous command is interrupted and the new command begins execution. The interrupted command's continuation then gets called with an error code of `:INTERRUPT`, allowing the robot program to decide how to handle the situation. Commands on different resources do not interfere with one another (explicitly, though subtle interactions may occur, eg, movement may cause a camera track to be lost).

3.1.3 Documentation example

In the remainder of this chapter, we will describe the various robot commands available in the simulator. Each command is summarized in a heading of the following form:

`camera-get-place-designator` *camera place-type continuation*
$\Rightarrow$ *camera desig*

This heading describes a command which uses the *camera* resource and has one other command argument, a *place-type*. The arguments to its continuation are the camera and a designator, after the standard error code (a keyword). So, one way to call this function would be:

```
(camera-get-place-designator (vref !>robot.cameras 1) 'door
                             (continuation (err cam desig)
                               (when err
                                 (out "Get designator failed on " cam
                                      " due to " err T))))
```

3.2 Geometry and Odometry

As discussed above, the simulated robot lives in a 2-dimensional world, and so has two directions of translation and one of rotation. Wherever (relative) positions are specified in the simulator, x increases to the right and y increases down (computer graphics convention). Angles are expressed in radians (`pi` and `2pi` are defined constants), with 0 angle pointing towards positive x and increasing clockwise.

3.2.1 Representing geometry

The basic datatype for 2D geometry programming that is provided is the `PT`, representing a 2D point with `x` and `y` slots. There are a number of Nisp operations for manipulating `PT`s (and other datatypes—see below). These operations are summarized in Table 3.1. A `PT` can be created in the usual way, by (`make PT` *x y*)

`add`	*pt1 pt2*	⇒ *pt*	Add two points (as vectors).
`add!`	*pt1 pt2*	⇒ *pt1*	Destructive add (change *pt1*).
`subtract`	*pt1 pt2*	⇒ *pt*	Subtract two points (as vectors).
`subtract!`	*pt1 pt2*	⇒ *pt1*	Destructive subtract (change *pt1*).
`neg`	*pt1*	⇒ *pt*	Negate a point.
`neg!`	*pt1*	⇒ *pt1*	Destructive negate (change *pt1*).
`mul`	*pt1 num*	⇒ *pt*	Multiply a point by a scalar.
`mul!`	*pt1 num*	⇒ *pt1*	Destructive multiply (change *pt1*).
`mul`	*pt1 num*	⇒ *pt*	Multiply a point by a scalar.
`mul!`	*pt1 num*	⇒ *pt1*	Destructive multiply (change *pt1*).
`dist`	*pt1 pt2*	⇒ *num*	Euclidean distance between points.
`norm`	*pt1*	⇒ *num*	2D Euclidean vector norm.
`equal`	*pt1 pt2*	⇒ *bool*	Are two points the same?
`copy-obj`	*pt1*	⇒ *pt*	Copy the point.

Table 3.1: Operations for manipulating points.

For representing uncertainty, we provide the `INTERVAL` datatype, with numeric slots `min` and `max`, representing the real interval from `min` to `max`. An interval is created by (`make INTERVAL` *min max*) For 2D, there is a `PT-INTERVAL` datatype, with slots `x` and `y`, each an interval along one axis. There are also computed slots `dx` and `dy` giving the widths of the interval along the corresponding axes. All of the point operations are extended to operate on intervals in the obvious fashion; there are also operations specific to intervals, summarized in Table 3.2. A point interval may be created by either `(make PT-INTERVAL` *x-int y-int*`)` or `(make PT-INTERVAL` *pt-min pt-max*`)`, where the min and max are opposite corners of the interval.

3.2.2 Odometry

The ARS MAGNA robot has built-in odometry which provides uncertain estimates of changes in position and direction over time. A robot has a set of *odometers*, each of which maintains an estimate of robot position and direction. It is suggested that different program modules use different odometers, to avoid introducing inconsistencies. When a robot is defined, it starts out with one odometer, which can be accessed by

`robot-main-odometer` *robot*

New odometers can be created by

`make ODOMETER` *robot name*

where *name* is a symbol giving a name to the new odometer; any odometer previously existing with *name* is destroyed. A particular odometer can be accessed by

`robot-odometer` *robot name*

A list of all a robot's odometers is returned by

`robot-odometers` *robot*

A program can read an odometer's current position/angle estimate(s) as well as reset the odometer to start again from zero. Resetting an odometer is done by calling one of the three functions

`reset-odometer` *odometer*
`reset-odometer-pos` *odometer*
`reset-odometer-ang` *odometer*

The first resets the odometer to zero; the other two reset the corresponding portion of the odometer. Position estimates are given as point intervals containing the possibilities for the actual relative position since the last reset. Angle estimates are similarly given as numeric intervals containing the possible relative angles. Current odometric estimates are returned by

`get-odometer-pos` *odometer* `&optional` *int*
`get-odometer-ang` *odometer* `&optional` *int*

The first returns the current odometric relative position estimate; the second the current relative angle estimate. If *int* is specified, it is modified and returned (for memory conservation).

3.3 Objects

As far as the robot is concerned, objects are described by a list of attribute-value pairs (the list is termed an *object descriptor*). This is represented as an association list (but for generality, values are stored in the `cadr`s of the list elements). Objects that match a given description can be looked for, and sensors can estimate the value of attributes for sensed objects. Note that equality of two sensed objects is not given and must be

`interval-intersect?`	*int1 int2*	⇒ *bool*	Do two intervals intersect?
`interval-subset?`	*int1 int2*	⇒ *bool*	Is *int1* a subset of *int2*?
`interval-intersect`	*int1 int2*	⇒ *int*	Return the intersection of two intervals (as an interval of the right type).
`interval-intersect!`	*int1 int2*	⇒ *int1*	Destructive intersection (modify *int1*).
`interval-width`	*int*	⇒ *num*	Width of an interval (largest distance between elements).
`interval-dist`	*int1 int2*	⇒ *num*	Minimum (axis-parallel) distance between interval edges.
`interval-overlap-width`	*int1 int2*	⇒ *num*	Width of the overlapping portions of two intervals.
`interval-lbi`	*int1 int2*	⇒ *int*	Least bounding interval (LBI) of two intervals (smallest interval containing both).
`interval-lbi!`	*int1 int2*	⇒ *int1*	Destructive LBI.
`add-to-lbi`	*int1 x*	⇒ *int*	Take the LBI of *int1* and *x* (of the appropriate type, a number or a point).
`add-to-lbi`	*int1 x*	⇒ *int*	Take the LBI of *int1* and *x* (of the appropriate type, a number or a point).
`add-to-lbi!`	*int1 x*	⇒ *int1*	Destructive version of the above.
`nominal`	*int1*	⇒ *x*	Center point of the interval.
`grow-interval`	*int1 num*	⇒ *int*	Grow the boundaries of an interval by *num* .
`grow-interval!`	*int1 num*	⇒ *int1*	Destructive interval growing.
`less`	*int1 int2*	⇒ *bool*	Is an interval entirely less than every element of another?
`greater`	*int1 int2*	⇒ *bool*	Is an interval entirely greater than every element of another?
`in-interval`	*x int1*	⇒ *bool*	Is *x* in the interval?
`interval-lift`	*x*	⇒ *int*	Return the interval containing only *x*
`lift-lbi`	*x1* [*x2* ...]	⇒ *int*	Return the smallest interval containing all *x*'s specified.

Table 3.2: Operations for manipulating intervals. Everywhere, *x* is an object of the appropriate type (number for intervals, point for point intervals).

Attribute	Values	Meaning
`:obsmove`	`{#t, #f}`	Does the object block movement?
`:obsview`	`{#t, #f}`	Does the object obstruct vision?
`:closed`	`{#t, #f}`	Is the object, if a container, closed?
`:color`	`{red, green, blue, violet, lightred, lightblue, lightgreen, lightviolet}`	What color is the object?
`:size`	*numeric*	How large is the object? Relevant to putting objects in containers, etc.
`:weight`	*numeric*	How heavy is the object? Relevant to picking things up, etc.
`:capacity`	*numeric*	How much can be put inside the object? A value of 0 indicates a non-container.
`:substance`	*numeric*	What substance parameter does the object have when 'viewed'? This will be seen directly if either the object obstructs vision or a camera is inside it; it will otherwise 'corrupt' measurements of wall substance behind it.

Table 3.3: Built-in object attributes.

determined by the robot program by analyzing object descriptions. Attribute names are Lisp keywords. Each attribute is either *symbolic* or *numeric*. Symbolic attributes can take on one of a predetermined set of values. There are a number of built-in attributes, some of which have special meaning to the system. World designers can add any other attributes they want. The built-in attributes are given in Table 3.3.

For example, a large, red box might have the following descriptor:

```
((:obsmove #f) (:obsview #f)  (:color red) (:substance 4.2)
 (:size 14)    (:capacity 13) (:weight 1))
```

A blue column might be described as:

```
((:obsmove #t) (:obsview #t) (:color blue) (:substance 7.3)
 (:size 10)    (:capacity 0) (:shape cylinder))
```

Here we see a user-defined attribute `:shape`. User defined attributes are described below, in Section 5.3.

3.4 Cameras and Perception

A robot may be equipped with a number of cameras, each of which is capable of long-range perception. As discussed above, there are two sorts of results of such perception, views and designators. Further, cameras are independently movable, both in terms of viewing direction (which also enables automatic object tracking) and in terms of being mounted on 'virtual' arms so that they can be used to look inside objects. Each camera has a *field of view*—an angular range defining (given the camera direction) the 2D wedge within which the camera can see; a camera also has a *resolution*—the number of samples (*pixels*) it uses when calculating a view. This section describes the commands for using simulator cameras. The robot's cameras are ordered, and can be accessed by an index (starting at 0) through the function `robot-camera`:

`robot-camera` *robot index*

3.4.1 Designators

The purpose of designators in the system is to provide a sort of indexical naming of objects in the system based on the notion of 'effective designators' developed in [7] (see also 'deictic representation' discussed in [1]). A designator provides a handle on an object or place (its *denotation*) that the system can use to control the robot, relative to the designator's denotation. In the simulator there are two distinct types of designators, *local* and *non-local.* Local designators denote objects 'next to' the robot, and package the information needed to directly manipulate the object (eg, to pick it up). Non-local designators denote objects or places further from the robot, and can only be used to go to the denoted location. There are two kinds of information stored in a designator that are accessible to the robot program. The first is an estimate of the direction (angle relative to the robot's current heading) of the designator's denotation. The second is a description of the thing denoted. For places, it is the place's type. For objects, it is a (partial) description of the object.

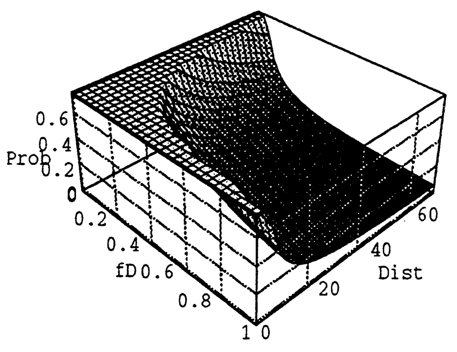
In the simulator, designators are structures of type `DESIGNATOR`. There are three accessible slots. `Type` is either `:local` or `:non-local` depending on the type of designator. `Angle` is the estimated relative angle (in radians) of the designator's denotation. Finally, `data` is a list, the description of the denotation. For places, the car of this slot is the place type. For objects, it is a list of attribute-value pairs, giving an estimated descriptor for the object. This descriptor is generated randomly for false designators.

3.4.2 Object matching

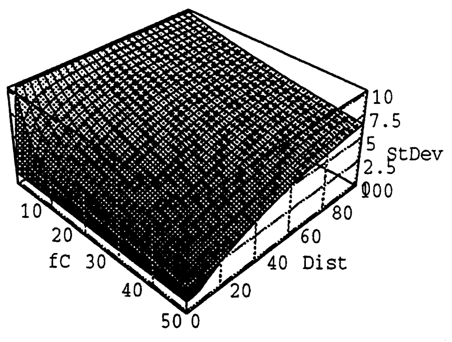
For acquisition[2] of object designators, a mechanism for matching object descriptors is needed. For more generality, since we will typically be matching user-specified descriptions to object descriptors, we match descriptors with *descriptor specifications* (given by the user). A descriptor specification is a list of attribute specifications given as follows. An attribute specification, generally, is a list whose `car` is the attribute specified and whose `cadr` is a value specification. A value specification for a discrete attribute is either the symbolic value desired, so `(:color red)` specifies objects whose color is red, or a list of possible values, as `(:color (green blue))` denoting objects that are either green or blue. A value specification for a numeric attribute can specify either a

[2]This should be 'acquisition and tracking', of course. However, at the present time, object matching is only done at designator acquisition time. This will be extended in the future.

particular value or a range (as a 2-element list (*min max*)); a matching threshhold may also be specified as the third element of the attribute specification. Since values will never match exactly, a default matching threshhold, θ, is used when none is specified. So, `(:size 4)` gives a size of 4, while `(:size (3 5))` specifies a size between 3 and 5, and `(:size 4 2)` specifies a size of 4 with a match threshhold of 2 (hence between 2 and 6), and finally, `(:size (3 5) 2)` specifies a size between 3 and 5 with a match threshhold of 2 (hence between 1 and 7). Matching is done with noise, so erroneous matching can occur.



(a)



(b)

Figure 3.1: Noise parameters as a function of distance. (a) Probability of a correct reading for a discrete attribute with base probability of 75%. (b) Standard deviation for a numeric attribute with a base of 1 and a maximum multiplier of 10.

Noisification

A descriptor specification matches a descriptor if the descriptor has a value for all attributes specified, and those values match the appropriate value specifications. Noisy value matching is done by first perturbing the actual value by noise and then checking to see if the perturbed value exactly fits the specification (if no matching threshhold is provided for numeric attributes, a default is used so that matching can occur). Noisification is different for discrete and numeric attributes. Each discrete attribute has a predefined base probability of correct readings, call it p. The actual noise probability is dependent on the distance d of the object from the camera, by the heuristic formula $p/\max(1, f_D d(1-p))$. Intuitively, as distance increases, so does the noise probability, but up to a maximum of p (see Figure 3.1a); f_D globally controls the point where this tops out. If a test on this probability fails (ie, the reading was corrupted), then the value 'seen' is chosen randomly from all possible values for the attribute (uniformly). Numeric attribute noisification is done using gaussian distributions about true. Given, for each numeric attribute, a base standard deviation σ and a maximum multiplier μ (ie, the maximum standard deviation is $\mu\sigma$), the actual standard deviation used when choosing a value is given by $\max(\sigma, \mu\sigma\frac{d}{d+f_C})$, increasing noise with distance. The steepness of increase is controlled by f_C (see Figure 3.1b).

Instantiated descriptors

When an object is sensed, a descriptor will be returned in a designator, giving the sensed readings. The attributes returned include those specified, as well as possibly some others; the probability of getting unspecified attributes depends on the attributes specified in the command. The more you specify top-down, the more useful information can be extracted from an image (or whatever base sensor data you have). Each attribute has a base probability of being sensed, so to speak, *de novo* (call it p). An attribute may also be said to be *supported* by some others, which means that information about a supporter gives information about the attribute. For example, `:capacity` is supported by both `:size` and `:weight`. Then, the actual probability of instantiating a non-specified attribute is also proportional to the number of supporters given in the descriptor specification, n. We use a simple binomial model, taking the probability as $1-(1-p)^{n+1}$. If the test for instantiation succeeds, a noisified value for the attribute (as above) is generated and added to the descriptor to be returned.

3.4.3 Camera commands

`camera-view` *camera continuation*
$\Rightarrow$ *camera view*

This function gets a view of the world from the given camera and returns it through a continuation. The view is a list of numbers, samples of the substance parameter of the nearest view obstruction, at equally-spaced angles in the camera's field of view. The length of the view is the resolution of the camera. Each sample (a *pixel*) is independently noisy. This function signals no errors.

`camera-get-place-designator` *camera place-type continuation*
$\Rightarrow$ *camera desig*
`camera-get-all-place-designators` *camera place-type continuation*
$\Rightarrow$ *camera desig-lst*

These use *camera* to get either one designator or a complete list of designators on a place of the named type, if any are visible in the field of view. It may also return a designator (or several) pointing to, essentially, nothing real. Place type designators are always non-local. Errors are as follows:

- `:ALREADY-TRACKING` *Camera* is already tracking another designator—tracking should be interrupted before attempting to look for something else.
- `:INVALID-PLACE-TYPE` *Place-type* is not the name of a place type.
- `:INVISIBLE-PLACE-TYPE` *Place-type* names a place type that can not be recognized at a distance.
- `:CAMERA-ENCLOSED` *Camera* is inside a container and so can't look for a place type.
- `:DESIGNATOR-NOT-FOUND` No designator was found for some reason.

`camera-get-object-designator` *camera desig-type obj-attrs continuation*
⇒ *camera desig*
`camera-get-all-object-designators` *camera desig-type obj-attrs continuation*
⇒ *camera desig-lst*

These functions look for objects in the field of view matching *obj-attrs*, a descriptor specification. The first returns a designator for a single such object; the second returns all those found (possibly missing some or including bogus designators). Finding all matching designators can take longer. Matching is done as described in Section 3.4.2; the designator returned contains a partially instantiated descriptor computed as described there. *Desig-type* is the type of designator desired, `:local` or `:non-local`. Errors possible are `:ALREADY-TRACKING` and `:DESIGNATOR-NOT-FOUND` as above.

`camera-track` *camera desig continuation*
⇒ *camera desig*

This sets *camera* to tracking *desig*. It will maintain the track by turning the camera if the robot or object moves. Note that to go to a designator, it must be tracked by a camera. While tracking, a local designator will automatically turn into a non-local one when appropriate (but the reverse is not true, since local designators are more powerful; thus information is lost). When a non-local designator is approached (by `robot-goto-designator`) and arrived at, the track will be lost, since when very nearby, objects and place types look very different than when far away. Errors are as follows:

`:ALREADY-TRACKING`	As above.
`:LOCAL-NOT-LOCAL`	*Desig* is a local designator, but the camera is not in the same place and so cannot track it. This may be due to the robot moving, but may also be due to inserting (removing) the camera into (out of) containers.
`:CAMERA-INSIDE`	The camera is inside a container and so can't track a non-local designator.
`:LOST-TRACK`	The track was lost for some reason. This can be due to occlusion, moving fast of the the field-of-view, or just random noise.

`camera-untrack` *camera continuation*
⇒ *camera*

Stops *camera* from tracking. The error `:NOT-TRACKING` is signalled if nothing is currently being tracked.

`camera-turn` *camera* Δ*angle continuation*
⇒ *camera* Δ*angle*
`camera-align-robot` *camera* *continuation*
⇒ *camera* Δ*angle*
`camera-align-place` *camera* *continuation*
⇒ *camera* Δ*angle place-type*

These functions all set the angle of *camera* and return an approximation to the relative angle moved. `Camera-turn` turns the camera by some relative angle. `Camera-align-robot` points the camera in the same direction as the robot. These two functions signal no errors (except for the standard `:INTERRUPT`). `Camera-align-place` points the camera in the direction of the nearest canonical direction for the current place type. If no place type is sensed, `:NO-PLACE-TYPE` is signalled.

`camera-insert` *camera* *desig continuation*
⇒ *camera desig*
`camera-remove` *camera* *continuation*
⇒ *camera*

These functions insert/remove *camera* into/out of a container, denoted by *desig. Desig* can either be a local designator or a storage bin, as usual. If removal is attempted when the camera is not inside anything, `:NOT-INSIDE` will be signalled. Insertion can cause the following errors:

`:BIN-CLOSED`	Insertion was attempted on a closed storage bin.
`:NOT-TRACKING`	*Camera* isn't tracking anything.
`:NOT-TRACKING-DESTINATION`	*Camera* isn't tracking the desired destination designator.
`:CAMERA-BUMP`	During the insertion process, *camera* bumped the destination object, losing the designator as well as failing to get inside.
`:INSUFFICIENT-CAPACITY`	The destination has no room for the camera.
`:NON-LOCAL-DESIGNATOR`	*Desig* is a non-local designator.

3.5 Robot Movement

`robot-set-speed` *robot* *new-speed continuation*
⇒ *robot speed*
`robot-set-turn-angle` *robot* *new-turn-angle continuation*
⇒ *robot turn-angle*

These two functions set the robots translation and rotation speeds, respectively. There are limits on these values (as world parameters) and the specified value will be truncated to fit them. `Robot-set-speed` signals either `:NEGATIVE-SPEED` or `:TOO-HIGH-SPEED` if *new-speed* is out of limits; `robot-set-turn-angle` signals either `:TOO-LOW-TURN-ANGLE` or `:TOO-HIGH-TURN-ANGLE` in analogous circumstances. The command returns the actual value that was set.

`robot-turn` *robot* Δ*angle continuation*
  $\Rightarrow$ *robot* Δ*angle*
`robot-about-face` *robot* *continuation*
  $\Rightarrow$ *robot* Δ*angle*
`robot-turn-left` *robot* *continuation*
  $\Rightarrow$ *robot* Δ*angle*
`robot-turn-right` *robot* *continuation*
  $\Rightarrow$ *robot* Δ*angle*

These functions turn the robot in place, returning a reading of the angular change. The basic function is `robot-turn` which turns the robot by (approximately) the given angle. The other three turn the robot by π, $-\pi/2$, and $\pi/2$, respectively.

`robot-align-place` *robot* *continuation*
  $\Rightarrow$ *robot* Δ*angle*

This function aligns the robot with its current place, if any, that is, it turns the robot to face in the nearest canonical direction. It may return an `:INCORRECT-PLACE-TYPE` error if it determines that the robot's current place type reading is invalid (but it may be wrong, of course). Otherwise it operates as the turning commands above.

`robot-turn-corner` *robot* *continuation*
  $\Rightarrow$ *robot* Δ*angle*

This command will turn the robot about a corner, either convex or concave. It does the intuitive thing. If it determines the robot is not at a corner, it signals the error `:NO-CORNER`; otherwise it acts as above.

`robot-go-forward` *robot* *time continuation*
  $\Rightarrow$ *robot*
`robot-goto-wall` *robot* *continuation*
  $\Rightarrow$ *robot*

These functions move the robot in a straight line forward, `robot-go-forward` for a fixed number of ticks, `robot-goto-wall` until a wall is encountered. Robot translation will signal `:ROBOT-BLOCKED` if the robot hits either a wall or an obstacle, and `:NOT-MOVING` if the robot's speed is 0. `Robot-goto-wall` will not signal `:ROBOT-BLOCKED` if the robot is determined to be at a wall (since in this case, the condition is a success).

`robot-goto-designator` *robot* *camera continuation*
  $\Rightarrow$ *robot camera desig*

This command tells the robot to try to get next to the thing (object or place) denoted by the designator tracked by *camera* (which designator is returned). The robot first tries to turn to face the designator, then tries to approach it by always moving towards it. This can cause problems if the designated object moves too fast, but that's life. The command succeeds when the robot is in the same cell as the denoted thing. False designators can also be 'tracked', but the robot will be randomly led down the garden path. In addition to the above-mentioned movement errors, the following errors will be signalled:

`:NOT-TRACKING`	*Camera* is not tracking anything. Note that this will also be signalled if the track is lost for some reason; this can be detected by the fact that the tracking process will signal an error.
`:GOTO-LOCAL`	The designator being used is local, so nothing need be done.

`robot-stop` *robot* *continuation*
$\Rightarrow$ *robot*

Stop *robot*. May take several ticks, depending on `braking*` (see Chapter 4).

3.6 Carrying Things

Simulator robots are (usually) equipped with a set of hands, each with a particular strength (governing how much it can carry). Hands can be used to pick things up, put them places, and carry them. The robot's hands are ordered, and can be accessed by indexing (starting at 0) through the function `robot-hand`:
`robot-hand` *robot index*
The robot has more carrying capacity through its storage bins, each of which has a limited capacity. Storage bins may be accessed by index, in a similar way to cameras and hands, by using
`robot-storage-bin` *robot index*

`hand-grasp-desig` *hand* *camera desig continuation*
$\Rightarrow$ *hand desig*

This commands *hand* to grasp the object denoted by a designator. When grasping, the hand must also be strong enough to hold the object. The grasping procedure may also bump the object, causing the designator to be subsequently lost (this happens silently). The following error conditions can occur:

`:HAND-GRASPING`	The hand is already holding something else.
`:DIFFERENT-LOCATION`	The hand and camera are in different locations (ie, at least one is inside a container the other isn't inside).
`:BAD-DESIGNATOR`	*Desig* is invalid.
`:NON-LOCAL-GRASP`	Tried to grasp something far away.
`:NOT-TRACKED`	*Camera* isn't tracking *desig*.
`:MISSED-GRASP`	Tried to grasp but missed for some reason.
`:INSUFFICIENT-STRENGTH`	Not strong enough to hold the object (it's too heavy).

`hand-ungrasp` *hand continuation*
⇒ *hand*

This tells *hand* to ungrasp the object currently held. The object is dropped into the current location of the hand, if it is inside a container (or storage bin). Designators on the object are unaffected (its trajectory can be followed). If the hand is empty, `:NOT-GRASPING` is signalled.

`hand-grasp-check` *hand continuation*
⇒ *hand*
`hand-track` *hand continuation*
⇒ *hand*

In order to keep track of what a hand is holding, the robot has virtual force sensors on each hand that can be tested by calling `hand-grasp-check`, which (when not interrupted) gives an error code of either `:GRASPING` or `:NOT-GRASPING`, depending on the hand's state. Hand status can be monitored over time by calling `hand-track`, which spawns a process that signals a `:GRASP-LOST` error if the hand loses its grip. Also, `hand-track` will be interrupted by any other commands to *hand* (signalling `:INTERRUPT`, as usual). If *hand* is empty to start, `:NOT-GRASPING` is signalled.

`hand-insert` *hand desig continuation*
⇒ *hand desig*
`hand-insert-bin` *hand storage-bin continuation*
⇒ *hand storage-bin*
`hand-remove` *hand continuation*
⇒ *hand*

These functions insert (remove) a hand into (from) a container. *Desig* may also be a storage bin; `hand-insert-bin` is provided as a useful special case. The following errors can occur:

`:BIN-CLOSED`	Tried to insert into a closed storage bin.
`:INSUFFICIENT-CAPACITY`	No room in the destination for the hand (hand size includes objects grasped).
`:HOLDING-THING`	Tried to insert the hand into the object it itself is holding.
`:HAND-BUMP`	Hand missed and bumped the destination object, also losing its designators.

`storage-bin-open` *storage-bin continuation*
⇒ *storage-bin*
`storage-bin-close` *storage-bin continuation*
⇒ *storage-bin jammed-part*

These commands open and close storage bins. This is useful to prevent objects from 'kicking' their way out of the bins. `Storage-bin-open` will signal `:BIN-OPEN` if *storage-bin* is already open. Similarly, `storage-bin-close` will signal `:BIN-CLOSED` if *storage-bin* is already closed. Also, if any robot-part (camera or hand) is inside the bin,

`:ROBOT-IN-BIN` will be signalled, and the offending part given by *jammed-part*. There is also a chance that an object in the bin will momentarily jam the bin door, resulting in a `:BIN-JAMMED` error.

3.7 Error Messages

`error-set` *-stuff-*
`warning-set` *-stuff-*
`message-set` *-stuff-*
`error-display` `&optional` *stream*

The simulator provides an error-message package for asynchronously displaying system error messages when the user wants them. Error and warning messages can also be given by the user. The idea is that messages are queued up and only printed when the system is explicitly told to do so; this also flushes the queue. The *type*-`set` functions take a list of items to be printed for a message of the appropriate type. `error-display` displays all messages on the queue to *stream* (default standard output), and flushes the queue. If `flush-all-errors*` is `T`, then no messages will be displayed; the default is `NIL`.

3.8 The Scheduler

Though you should never need to use it explicitly, we provide access to the simulator's underlying event scheduler just in case you think of something interesting to do that we didn't. The basic idea is that you can queue an *event* which is executed during a single tick. Note that a tick takes as long, in real time, as it takes to execute all the events in it. To get ongoing processes spanning multiple ticks, you need to queue a new event, chaining the original event. Each event has an associated resource; only one event is allowed at a time with a particular resource. Arbitration is handled by priority—events have an integer priority; when there is a conflict, the event with the higher priority gets queued (if priorities are equal, the event that got queued first wins). When an event gets bumped, it is said to have been *interrupted*, and an interrupt handler may be specified to be run in this case. An event may be queued to run at some given number of ticks in the future, as well. Events may be queued using the `schedule` special form:

```
(schedule (resource [ :priority priority ]
                    [ :interrupt inter ]
                    [ :when test ]
                    [ :timeout timeout ]
                    [ :timer time ])
     -body-)
```

This should be reasonably clear—*resource* is the resource for the event, *priority* is a number giving the priority of the event, *inter* is a Lisp expression to be evaluated if the event is preempted (note that the event itself, while running, can't actually be interrupted), and *-body-* is the code for the event. There are two ways to schedule an event for the future. The simplest is to specify *time*, the number of ticks in the future to run the event (by default the next tick). The event can also be conditioned on an arbitrary boolean expression, *test.* If a *test* is specified, then the event occurs when (and only if) *test* becomes true after at least *time* ticks and before *timeout* ticks. After *timeout* ticks, the event evaporates.

Chapter 4

Simulation Parameters

The physics of the simulated world can be adjusted by setting the various parameters that act as physical constants. Due to the abstract nature of the simulated environment, most of the parameters deal with the robot's relationship to the environment—sensing, effecting, and so forth. This section describes the various parameters that can be adjusted. One special kind of parameter are the *delay* parameters, which are used to simulate atomic actions (such as a hand insertion, which is a one step operation, internally) taking time, by scheduling the actual operation for some number of ticks in the future. Thus, if the world changes between-times, problems may (reasonably) occur.

The parameters are Lisp special variables; they are listed below with default values given in brackets.

Robot Movement

`Max-Speed*`	[1]	Maximum speed (in grid-cell units per tick) the robot can move. Setting this higher than 1 can cause strange effects (due to skipping over grid cells).
`Max-Turn-Angle*`	[$\pi/8$]	Maximum speed (in radians per tick) the robot can turn.
`Min-Turn-Angle*`	[0.05]	Minimum speed the robot can turn (due to friction and stuff).
`Braking*`	[2]	Number of ticks it takes for the robot to come to a stop. Currently, this is implemented as a simple delay then instantaneous stop.
`Motion-Noise*`	[0.1]	Percent noise in distance travelled as compared with desired speed (true distance is chosen from a uniform distribution about the desired speed). Also used to govern size of random odometric intervals returned.
`Robot-Turn-Max-Error*`	[0.01]	Percent noise in angle turned, similar to `motion-noise*`
`Robot-Align-Error-Prob*`	[0.1]	Probability that an align operation will 'suc-

		ceed' even if the wrong place type is specified.
`Robot-Move-Delay*`	[5]	Delay before a move command starts executing (inertia, friction).
`Robot-Turn-Delay*`	[2]	Delay before a turn command starts executing (inertia, friction).
`Robot-Align-Delay*`	[4]	Extra delay for an align operation to start (sensing, computation).

Cameras

`Camera-Angle-Max-Error*`	[0.02]	Max error (in radians) possible in the estimate of the angle of a designator.
`Scan-Density*`	[50]	Number of evenly-spaced samples taken in the field of view when scanning for a designator.
`Prob-Lose-Track*`	[0.01]	Probability that a track will be randomly lost on any given tick.
`Prob-Lose-False-Track*`	[0.4]	Probability that a track on a false designator will be randomly lost on any given tick. Higher than the above, since a false designator isn't real.
`False-Desig-Drift-Dist*`	[1]	Maximum distance that the perceived distance of a false designator will drift in a tick, while tracking.
`False-Desig-Drift-Ang*`	[0.2]	Maximum angle that the perceived distance of a false designator will drift in a tick, while tracking.
`Camera-Insert-Miss-Prob*`	[0.05]	Probability that an camera insertion will miss, also losing the designator.
`Camera-Insert-Lose-Desig-Prob*`	[0.1]	Probability that an insertion will lose the designator even when the insertion succeeds.
`Camera-Move-Delay*`	[2]	Time it takes to move a camera (for insertion).
`Camera-Align-Delay*`	[1]	Time it takes to turn a camera.
`Camera-Desig-Delay*`	[4]	Time it takes to acquire or start tracking a designator.

Sensor Error

`Type-False-Pos*`	[0.01]	Chance of sensing a (random) place type when at a non-typed location.
`Type-False-Neg*`	[0.05]	Chance of not sensing the current place type when one exists.
`Prob-Stop*`	[0.07]	When finding a false designator, a random angle in the field of view will be chosen, and the ray at that angle will be walked until either (a) the edge of the map is reached or (b) a test of probability `Prob-Stop*` succeeds. The point which is stopped at is the location of the false designator.
`Miss-Thing-Prob*`	[0.05]	Chance that an object simply will not be seen on

		a given scan, even if it would otherwise match a given descriptor specification.
`Miss-Place-Prob*`	[0.1]	Chance that a place simply will not be seen on a given scan, even if it would be seen otherwise.
`Pixel-Error-Range*`	[2.0]	Normal error range for pixel readings in views.
`Pixel-Outlier-Prob*`	[0.01]	Probability that a view pixel will be chosen uniformly from the entire possible range.
`No-Sensor-Error*`	[`nil`]	When set to `t`, this suppresses all sensing error.
`Discrete-Noise-Factor*`	[0.1]	In object matching, f_D. See section 3.4.2.
`Continuous-Noise-Factor*`	[0.05]	In object matching, f_C. See section 3.4.2.
`Standard-Match-Threshold*`	[0.5]	In object matching, θ. See section 3.4.2.

Hands

`Prob-Miss-Grasp*`	[0.04]	The probability that an attempted grasp will miss its object.
`Prob-Grasp-Lose-Desig*`	[0.1]	The probability that a grasp (even successful) will cause a designator on the object to be lost. This is tested independently for each designator on the object.
`Hand-Insert-Miss-Prob*`	[0.05]	Probability that a hand insertion operation will miss its destination.
`Hand-Insert-Lose-Desig-Prob*`	[0.1]	Probability than a missed hand insertion will result in a designator being lost.
`Hand-Move-Delay*`	[2]	Time it takes to move the hand.
`Hand-Grasp-Delay*`	[4]	Additional time it takes to perform a grasping operation.

Storage Bins

`Bin-Close-Jam-Prob*`	[0.01]	Probability that a storage bin will jam on closure, if there is anything in the bin to jam it. Note that bin jamming is an independent event; the fact that the bin is jammed now says nothing about it being jammed next tick.
`Bin-Delay*`	[2]	Time it takes to open or close a storage bin.

To enable the experimenter to easily examine behavior in a noiseless world, we provide the two functions:

```
clean-simulation
dirty-simulation
```

The first function sets all noise and error parameters to remove all noise, and the second restores the simulator to its state before `clean-simulation` was called.

Chapter 5

World Design

5.1 Maps

The datatype for environments (worlds) is `MAP`. Maps are created by the `defmap` special form:

```
(defmap name (width height &key cache)
        -map-clauses-
)
```

This defines a map named *name* which is *width*×*height* cells in size. *Cache* is the name of a file in which to cache the map for reloading, since building a map from scratch is quite time-consuming. Map clauses are forms which specify the walls and objects that should go in the map. The types of clauses are described below.

`(wall` *(x0 y0) (x1 y1) type param0* [*param1*]`)`
: This puts a straight line of filled cells in the map going from $(x0, y0)$ to $(x1, y1)$. The substance type of all the cells is *type* and the substance parameters are either all *param0* or are interpolated from *param0* to *param1* (if *param1* is specified.

`(rect` *(x0 y0) (x1 y1) type param0* [*param1 param2 param3*]`)`
: This puts a rectangle of filled cells with corners $(x0, y0)$ and $(x1, y1)$, with substance type *type* and either parameter *param0* or parameters interpolated from corner to corner between *param0* and *param1*, then *param1* to *param2*, and so on.

`(circle` *(x0 y0) rad type param*`)`
: This puts a (digitized) circle of filled cells with center (x, y) and radius *rad*, with substance type *type* and parameter *param*

`(obj` *name (x y) descriptor* `&rest` *contents*`)`

`(obj (`*name* `:prototype` *proto*`)` *(x y) descriptor* `&rest` *contents*`)`
: These forms create an object with name *name* at position (x, y). The first version simply creates an object with the given descriptor (*contents* is described below). The second form uses the descrip-

tor of the prototype object *proto* as the descriptor of the new object, with attributes specified in *descriptor* overriding those in *proto*. This allows common combinations of attributes to be packaged together. *Contents* described the contents of the object being defined, consisting of 0 or more object specifications, using a restricted version of the above syntax; one of:

(*name descriptor* `&rest` *contents*)
((*name* `:prototype` *proto*) *descriptor* `&rest` *contents*)

Note that contents can be specified recursively.

The following is an example of a simple map. This map defines an environment which is a single square room with a dividing wall. The room contains a lamp, a small box containing a pen, a large box containing a blue box (created based on a prototype box `box*`), and a table (created from `table*`).

```
(defmap big-room (75 75 :cache "~/cache/big-room")
        (rect (0 0)   (75 75)  Bricks  0   10   30   25)
        (wall (30 10) (30 50)  Slate   10  30)

        (obj lamp (20 20) ((:size 2) (:weight 1)
                           (:color lightred)
                           (:substance metal)))
        (obj small-box (20 20) ((:capacity 5) (:size 5) (:weight 2)
                                (:color lightblue)
                                (:substance wood))
             (pen ((:size .2) (:color green)))
             )
        (obj big-box (30 20) ((:capacity 11) (:obsview #t)
                              (:size 12) (:weight 4)
                              (:color red)
                              (:substance wood))
             ((a-box :prototype box*) ((:size 6) (:weight 3)
                                       (:color BLUE)
                                       (:substance wood)))
             )
        (obj (table :prototype table*) (40 15) ((:color violet)))
        )
```

5.1.1 Place types

Not only can you build different environments by placing walls and objects in various places, you can also design user-specified place types that can then be sensed and used for robot navigation. Recall that a place type is a stereotypical bit of local environment structure. In the simulator, this means specifying a particular set of local configurations of wall/empty cells. Place types are also associated with canonical directions, which the robot can align itself or its cameras with. A place type is defined using the `def-place-type` special form, explained below.

```
(def-place-type name (cell)
                [ :confusion confusion-prob ]
                [ :check-matchers ( {( match-form direction )}* ) ]
                [ :checker checker-expr ]
                [ :directions directions-expr ]
                [ :invisible invis ]
                [ :graphics ((-args-) -body- ) ]
)
```

The `def-place-type` form defines a place type named *name*. *Confusion-prob* gives the probability of finding a false place designator for the place type. The two canonical directions for a place of the given type are computed using *directions-expr*, an expression returning the two directions as multiple values. The variable *cell* is bound to the relevant cell during evaluation. The usual way this is done is to use the `canonical-dir` slot of the cell's `type` slot, which gives the cell's *basic canonical direction*. Given that, the canonical directions will usually be calculated as offsets from that basic value. If *invis* is `t`, designators cannot be acquired for the place type. A display method for cells of the given type may be specified in `:graphics`; the use of this clause is described in Appendix B.1.

There are two ways of specifying the wall configurations defining a given place type. The simpler is by giving a set of *match-forms* with associated canonical *directions*. These specify when a each possible cell of interest should be classified as being of the new place type. Each match form is a list of lists, representing the cells surrounding the cell of interest, which is in the center of the matrix. The entries are one of: `nil` (empty), `t` (wall), or `?()` (don't care). They may also be abbreviated as `-`, `*`, and `x`, respectively. Thus, any cell in a map whose surrounding cells have the corresponding wall structure will be classified to be of type *name*. Further, each such cell is associated with the basic canonical direction given as *direction*. An example of a place type definition using match forms is given below. This example defines a doorway place type, for narrow openings in walls. Each of the four match forms describes a doorway configuration in a different direction (horizontal, vertical, and two diagonals), giving one of the directions straight through the doorway as a basic canonical direction. The two canonical directions for a doorway are given as the two opposite directions looking through the doorway.

```
(def-place-type doorway (cell)
  :confusion 0.05
  :check-matchers ( ( ((nil nil nil)
                       ( t  nil  t)
                       (nil nil nil)) (/ pi 2) )
                    ( ((nil  t  nil)
                       (nil nil nil)
                       (nil  t  nil)) 0.0 )
                    ( (( t  nil nil)
                       (nil nil nil)
                       (nil nil  t )) (/ pi 4) )
                    ( ((nil nil  t )
                       (nil nil nil)
                       ( t  nil nil)) (* pi 3/4) )
                    )
  :directions (values !>cell.type.canonical-dir
                      (+ pi !>cell.type.canonical-dir))
  :graphics ((win cx cy)
             (xlib:draw-rectangle win draw-gcontext*
                                  (1+ cx) (1+ cy)
                                  (- graphics-scale* 2) (- graphics-scale* 2)
                                  nil)
             ))
```

A more complex, but more flexible, way of specifying place type structure is by specifying, instead of a set of check matchers, *checker-expr*. This is an expression that will be evaluated with *cell* bound to each map cell, whose return values determine place type classification relative to the new type. The expression should return four multiple values: the first is a boolean, specifying whether a place type classification should be performed, the second and third are integer offsets from *cell*, specifying which cell should be so classified (ie, *cell* is not necessarily the cell to classify), and the fourth gives the basic canonical direction for the classified cell. A useful function for place type checkers is:

`cell-occupancy-window` *cell width height*

This returns a *width*×*height* occupancy matrix (`t` = wall, `nil` = empty) centered on *cell* which can then be matched against for place type determination. An example of the definition of a convex corner place type, using a checker expression, is given below. The checker finds filled cells with walls extending in two perpendicular directions with empty spaces on the convex 'outside'. In such situations, it classifies the corner empty cell, giving as canonical directions the two perpendicular directions away from the walls.

```
(def-place-type convex (cell)
  :confusion 0.03
  :checker (if !>cell.filled?
               (let ((win (cell-occupancy-window cell 3 3)))
                 (cond ((matchq ((nil nil nil)
                                 ( t   t  nil)
                                 (?()  t  nil)) win )
                         (values t 1 -1 0.0))
                        ((matchq ((?()  t  nil)
                                  ( t   t  nil)
                                  (nil nil nil)) win )
                         (values t 1 1 (/ pi 2)))
                        ((matchq ((nil nil nil)
                                  (nil  t   t)
                                  (nil  t  ?())) win )
                         (values t -1 -1 (* 3/2 pi)))
                        ((matchq ((nil  t  ?())
                                  (nil  t   t)
                                  (nil nil nil)) win )
                         (values t -1 1 pi))
                        (t (values #f 0 0 0))))
               (values #f 0 0 0))
  :directions (values !>cell.type.canonical-dir
                      (- !>cell.type.canonical-dir (/ pi 2)))
  :graphics ((win cx cy)
             (xlib:draw-arc win draw-gcontext*
                            cx cy
                            graphics-scale* graphics-scale*
                            0 (* 2 pi) nil)
             ))
```

5.2 Robots

Robots are of type `ROBOT` (surprise!) and have no user-accessible slots. They are defined using the **defrobot** special form:

(defrobot *name* (*environment* (*x y*) *ang*)
        [:cameras ({(*fov res*)}*)]
        [:hands ({*strength*}*)]
        [:bins ({*capacity*}*)]
)

Name is bound as a special variable to the robot structure. The robot is placed in the *environment* at position (x,y) pointing in direction *ang*. The robot has a set of cameras, each specified by its field of view *fov* and resolution (number of view pixels) *res*. Hands are specified by their *strength* and storage bins by their *capacity*. For example:

```
(defrobot robbie (watson (10 5) 0)
         :cameras ( ((/ pi 4) 3) ((/ pi 2) 5) )
         :hands (10 4)
         :bins (20 5)
         )
```

defines a robot named `robbie` who lives in the `watson` world (defined elsewhere). `Robbie` has two cameras, one with 45° of view and 3 view pixels, the other with 90° and 5 pixels, respectively. It has two hands, capable of picking up weights of 10 and 4 weight units, and two storage bins, with capacities of 20 and 5 size units.

5.3 Things

Above, in the section on map definition, we saw one way to create objects, by placing them a priori in a world map. They can also be created on the fly, using `new-thing`:

```
(new-thing name { loc | (:with obj) | (map x y)}
                descriptor
                [ :prototype proto ]
         )
```

This creates an object named *name* (recall that object names are only for convenience and need not be unique). The location of the new object is either inside *loc* (a container, hand, or storage bin), at the same location as *obj*, or at position (x, y) in *map*. If *proto* is specified, it is used as a prototype for the new thing, that is, *proto*'s attributes and contents are copied into the new thing and then modified by *descriptor*. Otherwise, *descriptor* is the complete descriptor of the thing. The thing is returned from `thing`, primarily for use in defining other things. A common use of this is for creating prototype objects, with location `nil`, which are only used for packaging up useful combinations of attributes. For example, a box prototype might be specified as:

```
(defvar box* (new-thing box nil ((:capacity 8) (:size 8) (:weight 2))))
```

defining a 'generic' box.

Some other useful functions are those that manipulate object attributes. These will be most useful when defining kickers or manipulations (see below). The basic attribute accessor is

`get-attribute` *thing attribute*

which returns the current value of *attribute* for *thing* (`nil` if not specified). This function is settable, or you can use

`set-attribute` *thing attribute new-value* to set an attribute's value. An entire set of attributes can be changed by using

`change-attribute` *thing descriptor*

where *descriptor* is a thing descriptor giving the new values for various attributes.

5.3.1 Kickers

While the variety of objects possible in our system can make for interesting static environments, the true power of a robotic planner is its ability to deal with a dynamic world. Therefore, the simulator contains a mechanism for automatic environmental dynamism. The basic idea is that objects move (or change) every so often. This idea is embodied in the notion of a *kicker*, a piece of Lisp code attached to an object which is executed at random intervals. Kickers are generally used to randomly move objects, thus (second-order) simulating the effects of other agents in the environment. Kickers can also change object attributes, or create/destroy objects. The basic way to define a kicker is by the `defkicker` special form:

```
(defkicker name (var)
           ( [ :prob prob ]
             [ :test test-form ] )
  -body- )
```

This defines a kicker named *name* that will execute *-body-* with probability *prob* on any tick when *test-form* evaluates to true. For both *test-form* and *-body-*, *var* is bound to the kicker's object. `Defkicker` thus defines an abstract kicker that can be attached to any object through the use of (`set-kicker` *thing kicker-name*). A one-of-a-kind, idiosyncratic kicker can be defined for an object using the `put-kicker` special form:

```
(put-kicker name (var thing)
            ( [ :prob prob ]
              [ :test test-form ] )
  -body- )
```

It is nearly identical to `defkicker`, except that *thing* specifies the particular object that the kicker is to be assigned to, and *name* is specified only so that the kicker can be redefined—it is not globally bound to the kicker (for use by `set-kicker`). This can also be used to remove a kicker with a given name (even a global one) from an object, by specifying a null body.

Since the body of a kicker is only executed during one tick, if longer-term actions are required (for example, moving an object along a trajectory), the kicker must be rescheduled. This is done using the function `run-kicker`, whose single argument should be specified as `self` (which is bound to the kicker currently being executed). Moving an object is accomplished using the function `kick-thing`:

`kick-thing` *thing* Δx Δy *kick-if-wall*

This function will move *thing* by $(\Delta x, \Delta y)$. If the destination is inside a wall and *kick-if-wall* is true, a non-wall neighbor of the wall cell will be found and *thing* will be kicked there instead. If *kick-if-wall* is false, however, kicking into a wall will do nothing, and the function will return `nil` (otherwise, on success, it returns `t`). Similarly, if the object is in a closed object or storage bin, the kick will fail and `kick-thing` will return `nil` (regardless of the value of *kick-if-wall*).

5.4 Defining new attributes

Since it is impossible for us to predict all possible types of environments that users of the simulator may want to model, and in particular, what types of objects those environments contain, we have also provided a facility for defining new object attributes, which can then be used in descriptor specification and can be sensed, in just the same way as built-in attributes. An attribute is defined by specifying two things: its domain (the possible values it can take on) and its sensing parameters (please see section 3.4.2 for a detailed discussion of the meaning of these parameters). An attribute is defined by the `defattribute` special form:

```
(defattribute name domain
                 [ :success success ]
                 [ :base base ]
                 [ :max-mult max-mult ]
                 [ :noticed noticed ]
                 [ :supported-by ({supported-by}*) ]
)
```

This defines an attribute named *name* (by convention a keyword). If *domain* (which is unevaluated) is a list, the elements of the list are the possible values of the attribute, and it is declared to be discrete. To specify a numeric attribute, *domain* should be the symbol `NUMERIC`. For a discrete attribute, *success* is the basic probability of a correct reading of the attribute's value (modified by distance as above). The default success probability is 1. Numeric attributes have two sensor noise parameters: *base* giving the basic standard deviation of sensor readings (default 0) and *max-mult* giving the maximum standard deviation multiplier with increasing distance (default 1). *Noticed* gives the base probability of the attribute being sensed when not specified by the robot. Each *supported-by* is the name of an attribute which when specified by the robot increases the chances of noticing the value of the new attribute.

For example, the `:color` attribute could be defined by:

```
(defattribute :color (red green blue violet
                      lightred lightblue lightgreen lightviolet)
          :success .4 :support .4 :supported-by (:substance))
```

This specifies that color is a discrete attribute with various particular colors possible, that color is not likely to be sensed correctly (due to lighting difficulties, etc.), that it is not likely to be noticed, either, and that noticing color is more likely when examining substance since when the material is known, the color is more likely to be known. Size, on the other hand, would be defined as:

```
(defattribute :size NUMERIC
              :base .8 :max-mult 4
              :support .7 :supported-by (:capacity :obsview :weight))
```

Size is thus a numeric parameter, with a reasonably small margin of error and a good chance of being noticed; size is also more likely to be noticed when trying to compute an object's carrying capacity, weight, or whether it obstructs the robot's view.

5.5 Manipulations

A final way to enhance a simulated environment is to define *manipulation methods*, by which the robot program can manipulate objects in the environment. For example, a manipulation could be defined to open containers by setting the value of their `:closed` attribute. A given manipulation may have a number of different methods, one of which is applied in a given case based on the particular objects passed to it. This provides a object-oriented sort of system, based on pattern-matching on descriptors. Also, since a manipulation is a robot action, it may have a probability of failure, as well as a delay before it actually occurs.

5.5.1 Defining manipulations

A manipulation method is defined by **defmanipulation**:

```
(defmanipulation name (-object-params- :by resource-param -other-params-)
                 ( [ :success succ ]
                   [ :delay delay ] )
  -body- )
```

This defines a manipulation method for the manipulation *name* with success probability *succ* (default 1) and delay *delay* (default 0). The parameter list has several components which will be discussed one by one. First, *-object-params-* lists the object parameters to the function. Each variable name must be followed by a dash (-) followed by an extended descriptor specifier, which specifies which objects can be bound to the variable. The extended specifier is just like a normal descriptor specifier (as described in section 3.4.2), with the extension that values can be given as Nisp matchvars (eg, `?x`) instead of actual values. In the first descriptor in which the matchvar appears, when matched it is bound to the value of the applicable attribute. When the same matchvar appears later in an attribute specification for another object parameter, the value of that attribute must match the matchvar's binding. This is made clearer by the example further below. The *resource-param* and *-other-params-* are specified as ordinary function parameters; the only special thing about the *resource-param* is that when the manipulation method is run, the value of the *resource-param* is used as the scheduler resource for the manipulation event. The *-body-* should return two values, the first an error code (as above) or `nil` for successful execution, the second a return value to be passed to a continuation.

An example of a definition for a lock manipulation is as follows.

```
(defmanipulation lock (container - ((:capacity (0.1 9999)) (:lock-id ?x))
                        key - ((:type :key) (:lock-id ?x))
                        :by hand - HAND)
      (:success .9  :delay 1)
  (cond ((in-location key hand)
         (setf (get-attribute container :locked) t)
         (values nil hand))
        (t
         (values :NOT-HOLDING-KEY hand))))
```

This defines a manipulation method which locks a container when using a key with the same `:lock-id` as the container. The method usually works when applicable (90% of the time). If the hand is not holding the key, an error is signalled. Other methods could also be defined, for locking doors or bicycles, say.

5.5.2 Running manipulations

When executing a manipulation from a robot program, the object parameters are specified by local designators which (with luck) denote the desired objects of the action. A manipulation is generally executed via the **`manipulate`** special form.

`manipulate` *manipulation-name -args- continuation*

The arguments are as specified in the manipulation method definition(s), with designators representing the objects they denote. So, if **`box-desig`** is thought to denote a box, **`key-desig`** a key held by **`right-hand`**, and **`cont`** is a desired continuation (see below), the box may (perhaps) be locked by

`(manipulate lock box-desig key-desig hand cont)`

Note that `:by` is not used here. For dealing with return values, manipulations use continuations in the same way as built-in robot commands. Manipulation continuations take two arguments. The first, as usual, is an error code. The one standard manipulation error is `:FAILED`, which is signalled when either no applicable method is found or an applicable method fails its success probability test. Manipulations may also define their own error codes. The second continuation argument is a manipulation-dependent return value. If the user wants to decide which manipulation to run at run time, she may, provided the parameter lists of the possible manipulations are similar enough. This may be done by using the function **`run-manipulation`**:

`run-manipulation` *manipulation-name -args- continuation*

So, using this, the above lock example could be done as

`(run-manipulation 'lock box-desig key-desig hand cont)`

Bibliography

[1] Philip E. Agre. *The Dynamic Structure of Everyday Life.* PhD thesis, MIT Artificial Intelligence Laboratory, 1988.

[2] Badr Al-Badr and Steve Hanks. Critiquing the Tileworld: Agent architectures, planning benchmarks, and experimental methodology. *AI Magazine*, 1991. submitted.

[3] Sean P. Engelson and Drew V. McDermott. Image signatures for place recognition and map construction. In *Proceedings of SPIE Symposium on Intelligent Robotic Systems, Sensor Fusion IV*, 1991.

[4] R. James Firby. *Adaptive Execution in Complex Dynamic Worlds.* PhD thesis, Yale University, January 1989. Technical Report 672.

[5] R. James Firby and Steve Hanks. The simulator manual. Technical Report YALEU/DCS/TR-563, Yale University Department of Computer Science, 1987.

[6] Drew McDermott. Revised NISP manual. Technical Report 642, Yale University Department of Computer Science, 1988.

[7] Drew McDermott. A reactive plan language. Technical Report 864, Yale University Department of Computer Science, 1991.

[8] A. Philips and J. Bresina. NASA Tileworld manual. Technical Report TR-FIA-91-11, NASA Ames Research Center, Code FIA, 1991.

[9] Martha E. Pollack and Marc Ringuette. Introducing the Tileworld: Experimentally evaluating agent architectures. In *Proc. National Conference on Artificial Intelligence*, 1990.

[10] Robert W. Scheifler et al. *CLX Common LISP X Interface*, 1988,1989.

[11] A.L. Samuel. Some studies in machine learning using the game of checkers. In *Computers and thought.* McGraw-Hill, 1963. Also in IBM Journal of Research and Development (1959).

[12] Steven Vere and Timothy Bickmore. A basic agent. *Computational Intelligence*, 6(1), Feb 1990.

Appendix A

Running the Simulator

The simulator is loaded into Nisp by typing `(dsklap "`*ars-dir*`/ars-magna")`, where *ars-dir* is the directory in which the simulator has been installed[1]. This will also define a NISP logical name `ARS`, bound to the installation directory. Once the system is loaded, a robot can be started by typing `(simulator` *robot*`)`, which will pop up a graphic display of the robot and its world (see Appendix B). There is a command-line interface, described below, that you can run by typing `(cli)`.

If you have any problems, please send email to `engelson@cs.yale.edu`.

[1]At Yale as of this writing, this directory is `/cs/yale/src/lisp/nisp/ars-magna` on Thailand.

Appendix B

The Graphical Display

This appendix describes ARS MAGNA's X-based graphical display system, activated when `simulator` is called. It is not currently a full interface (there is no input), but the various displays allow the user to see concisely what is going on in the world. There are several windows showing different aspects of the world—there is a window showing: the world map, the robot's cameras, its hands, and its storage bins. The remainder of this section explains the various types of displays. It may be helpful to run the system and refer to an actual display. Although the figures here are in black and white, color is used when available; details are described below.

The world

The main display window shows a bird's eye view of the robot and its world. Figure B.1 shows an example of this display. Solid black areas are walls, white areas are open space. In color, walls are displayed in shades of gray, denoting different substance parameters. At the upper left is the robot, shown as a circle containing an arrowhead. The arrowhead points towards the robot's current direction. Not visible in this picture are small radial lines, showing the directions of the robot's cameras (they are more visible in color). Directly before the robot are a couple of objects, shown as small vertical lines. In color, different objects are shown as differently-colored lines, so they can be more easily distinguished. Cells with a visible place type (see Section 5.1.1) are marked, depending on the particular type. In black-and-white the marks are symbols, here you can see a small circle denotes a convex corner; a dot, a concave corner or beak; and a square, a doorway. On a color display, the different place types are indicated by filled cells of different colors: convex = pink, concave = light pink, doorway = light blue, and beak = violet. User-defined place types may also have display methods of this sort if given a `:graphics` method, as described further below. The scale of the world display can be changed by setting the variable `graphics-scale*`, which specifies how many pixels there are to a grid cell side.

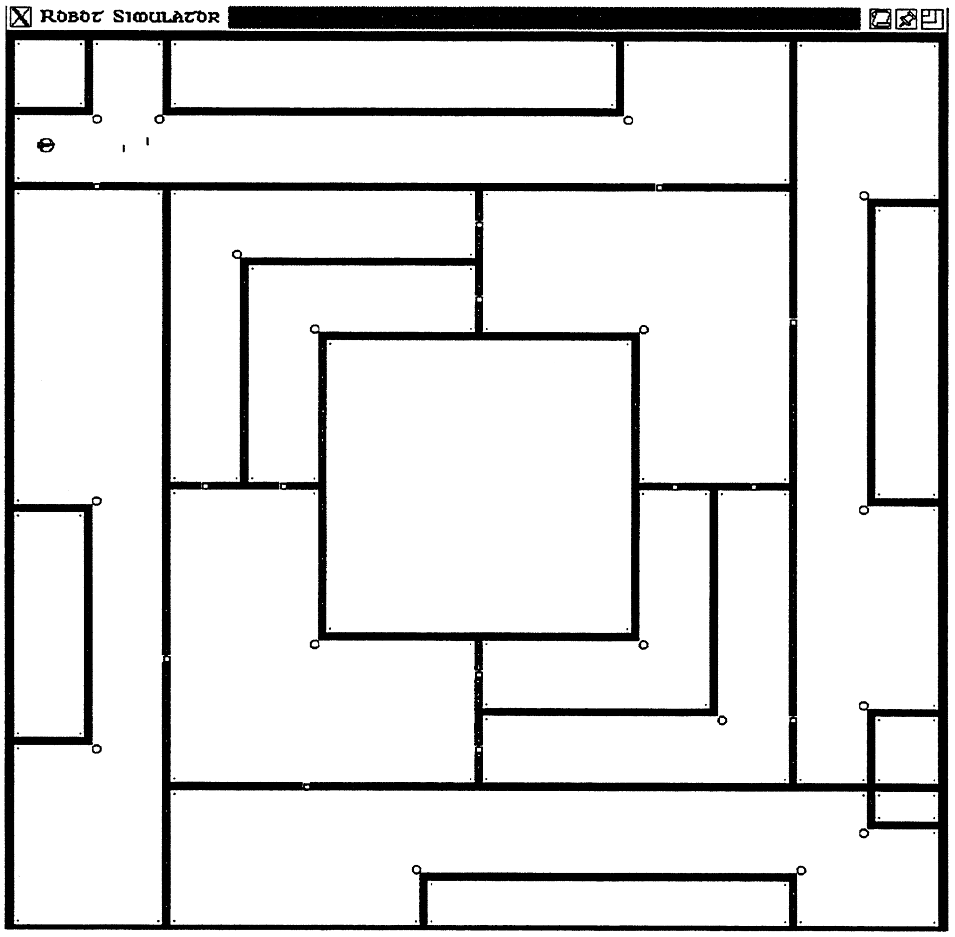
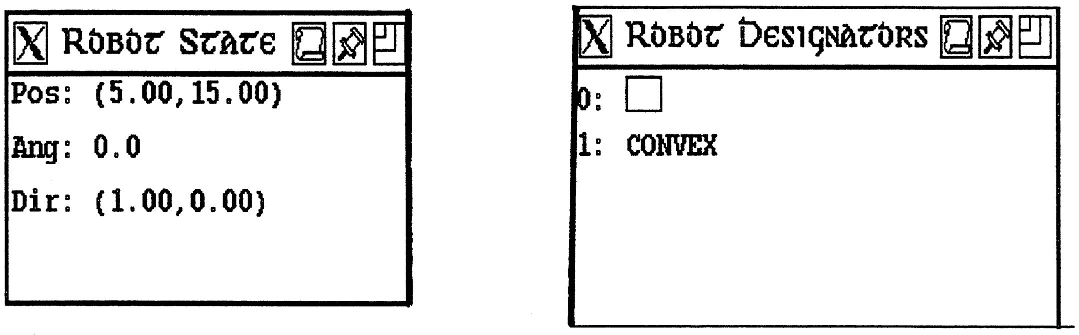


Figure B.1: The Ars Magna world display. The robot is at the upper left.

Objects

As mentioned above, objects appear in the world map as small vertical lines in particular grid cells. In the more detailed displays, objects are shown more completely. If an object has a value for the attribute `:draw-function`, which should be a procedure taking as arguments an XLIB window, the object, x and y positions, and a width and height (within which to draw the object), that procedure will be used to draw a representation of the object where needed. Otherwise, by default, each object is drawn as a rectangle, of width proportional to its `:size` value (see Figure B.2(b) for an example). If the object obstructs view, an 'X' is drawn over it, if it obstructs movement, it is drawn as a solid rectangle. On color displays, the `:color` attribute is used to determine what color the object is drawn with.

Robot state



(a) Robot State

```
Pos: (5.00,15.00)
Ang: 0.0
Dir: (1.00,0.00)
```

(b) Robot Designators

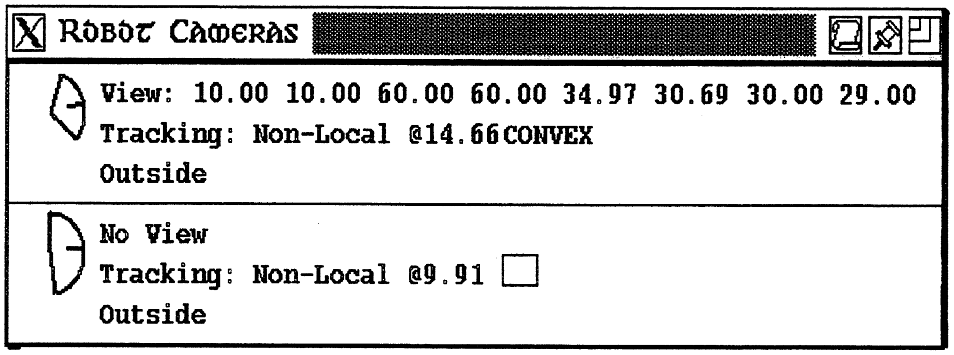
```
0: □
1: CONVEX
```

Figure B.2: The Ars Magna robot status displays. (a) Robot state. (b) Robot designators.

Figure B.2 shows two displays of generic robot status information. The "Robot state" window shows the position and direction of the robot. The "Robot designators" window shows a list of the robot's current designators. Designators are displayed by their denotations. Objects are drawn as described above; place types' names are shown, and false designators are noted by "FALSE". Since the list of designators can grow large, the list can be scrolled up and down by clicking the right and left mouse buttons, respectively.

Cameras

The robot's cameras are displayed in detail in their own window, as shown in Figure B.3. At the left of each camera display is a depiction of the camera as a wedge; the direction of the wedge is the direction of the camera, and its width is the camera's field



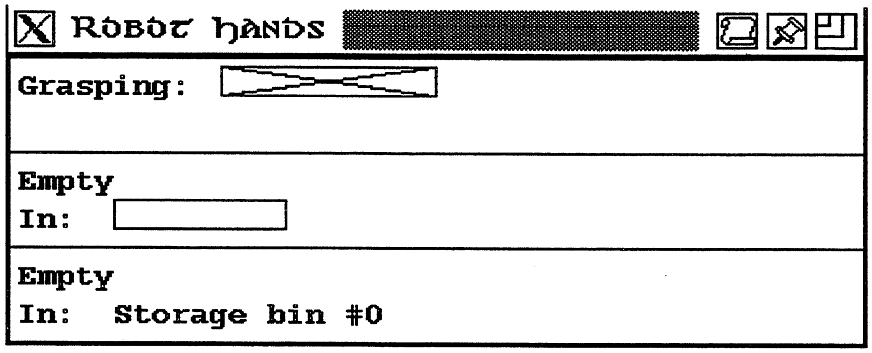
View: 10.00 10.00 60.00 60.00 34.97 30.69 30.00 29.00
Tracking: Non-Local @14.66CONVEX
Outside

No View
Tracking: Non-Local @9.91
Outside

Figure B.3: The ARS MAGNA robot cameras display.

of view. If the camera is tracking a designator, the estimated designator direction is shown as a tick in the field of view (as shown). On the right are three lines showing: the camera's view, its tracked designator, and its containment. The view is shown as a list of numbers. If the camera is tracking, the designator's type, estimated distance, and denotation are shown. If the camera is inside a container, that fact is shown in the third line of the display (in the same manner as shown below, in Figure B.4).

Hands



Grasping:

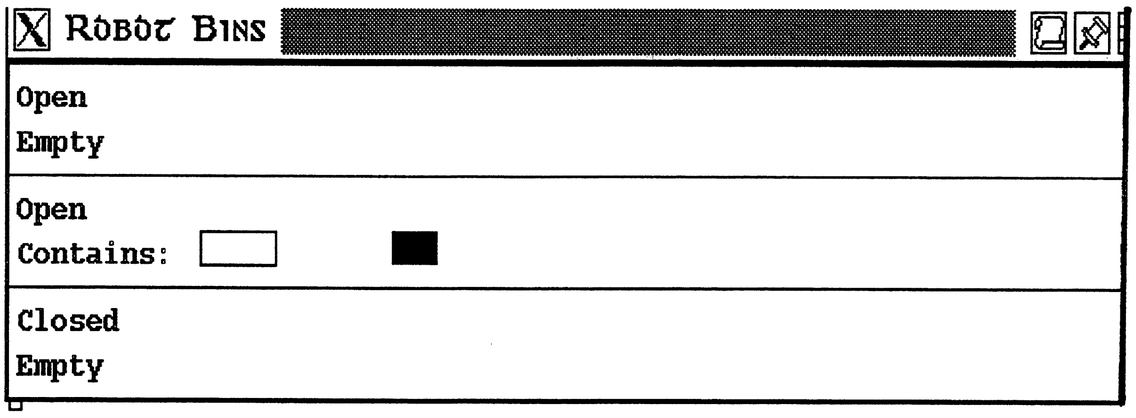
Empty
In:

Empty
In: Storage bin #0

Figure B.4: The ARS MAGNA robot hands display.

Two pieces of information are displayed for each of the robots hands, as depicted in Figure B.4: the hand contents, and its containment. If the hand is grasping an object, that object is displayed (in the manner described above). If the hand is inside a container or storage bin, that fact is also shown, by either drawing the object, or printing which storage bin the hand is inside.

Storage bins



Open
Empty

Open
Contains:

Closed
Empty

Figure B.5: The ARS MAGNA robot storage bins display.

Two data about storage bins are shown in the storage bin window: whether each bin is open or closed, and what it contains. As shown in Figure B.5, closure status is noted simply, by text. Bin contents are shown as a list of objects. Note that robot parts inside bins are not shown (since they take up no 'room').

B.1 Place type graphics methods

Recall that when we discussed defining place types in Section 5.1.1, `def-place-type` had a mysterious keyword argument `:graphics`. This argument allows the place type designer to specify how cells with that place type should be displayed. This interface is somewhat primitive at present, but not too difficult to use. Recall `def-place-type`:

```
(def-place-type name (cell)
                -other-keywords-
                [ :graphics ((winvar xvar yvar) -body- ) ]
)
```

The `:graphics` clause specifies a funxtion for drawing the place type's representation, using the CLX interface to X windows [10]. The first sublist specifies variable names to be bound when executing the body: *winvar* is bound to the window to draw into, and *xvar* and *yvar* to the position (in pixels) of the cell to be drawn. There are three important global variables to be used here, `graphics-scale*`, described above, `draw-gcontext*` (in the NISP package, as are all variables here *except* `graphics-scale*`), whose value is a CLX graphics context to be used in drawing operations, and `in-color*` (also in the NISP package), which is `#T` if the current display is in color and `#F` otherwise. The variables `black*` and `white*` are bound to their respective colors, and `grey*` is a 64-vector of shades of grey (0 is black, 63 is white). There are also several variables whose values are 12-vectors of colors, giving a range from dark to light shades of particular hues: `red*`, `green*`, `blue*`, `violet*`, `lightred*`, `lightgreen*`, `lightblue*`, and `lightviolet*`. A particular color can be easily selected for drawing by using the macro: `(with-color` *color -body-*`)`; if the display is not in color, black is used. Similarly, a particular line style and/or width may be selected (see the CLX manual for available styles) by using `(with-lines ([:width` *w*`] [:style` *st*`])` *-body-*`)`.

The use of `:graphics` should be made clearer by reference to the following example, taken from the definition of the convex corner place type:

```
:graphics ((win cx cy)
           (if in-color*
               (with-color (vref lightred* 9)
                 (xlib:draw-rectangle win draw-gcontext*
                                      cx cy
                                      graphics-scale* graphics-scale*
                                      t))
               (xlib:draw-arc win draw-gcontext*
                              cx cy
                              graphics-scale* graphics-scale*
                              0 (* 2 pi) nil)))
```

This bit of code does the following. If the display is in color, a filled square of side `graphics-scale*` with origin at (x,y) (covering the cell) is drawn in pink (light red). Otherwise, a circle is drawn, inscribed in that same square. Care should be taken not to draw outside this square, since unpredictable things may happen then.

Appendix C

RPL Interface

RPL [7] is a robot programming language designed for automatic development of reactive plans. This appendix describes an interface to Ars Magna from RPL. We generally assume familiarity with RPL (please see [7] for details). One important class of RPL entities which bears mentioning here is the *fluent*. Fluents are time-varying quantities, by which dynamic behavior of a program/plan can by controlled. For example, `(wait-for` *f*`)` waits until the fluent *f* becomes true. This locution (and others) leads to the idiom used here of having fluents which are *pulsed* (set to true for an instant) to signal particular conditions.

All interface functions operate on one particular robot (making plans much more concise). Cameras, hands, and storage bins are referred to by integer indices given by the order they appear in the `defrobot` form (starting at 0). The functions are of two types: commands and accessors. Commands spawn a process in the simulator to perform some action or sensing, and return results via RPL fluents. Some fluents relating to cameras, hands, or storage bins come in vectors, one for each corresponding robot part. Accessors allow a plan to determine the configuration of the robot that is being used, for example the number of cameras mounted. All RPL interface functions are in the `NISP` package and are prefixed by "`rpl-`"; the names are otherwise generally the same as the corresponding regular Ars Magna functions. All arguments can also be given as fluents with values of the appropriate types.

Initialization

`rpl-init-robot` *robot*
This initializes the RPL interface to use *robot*. This must be called before doing anything else.

Robot fluents

`robot-stopped*` Pulsed when the robot stops moving, whether or not a movement was successfully completed.

`robot-movement-error*` The error code signalled by the last command to complete (or be interrupted).

`robot-angle-diff*` Set to the estimated change in angle over a turning operation.

`robot-arrived-at*` Set to a designator that the robot successfully moved to.

`robot-place-input*` Pulsed when the robot's place type is sensed.

`robot-place*` Set to the last robot place type sensed.

Robot functions

`rpl-robot-speed`
`rpl-robot-turn-angle`
Return the robot's current speed and turning angle, respectively.

`rpl-robot-num-cameras`
`rpl-robot-num-hands`
`rpl-robot-num-bins`
Return the number of cameras, hands, or storage bins mounted on the robot.

`rpl-robot-set-speed` *speed*
`rpl-robot-set-turn-angle` *angle*
These two commands set the robot's speed and turn angle, respectively. Any error code is put into `robot-movement-error*`.

`rpl-robot-turn` *angle*
`rpl-robot-about-face`
`rpl-robot-turn-left`
`rpl-robot-turn-right`
`rpl-robot-turn-corner`
`rpl-robot-align-place`
These commands turn the robot in various ways. When the turn is completed, `robot-stopped*` is pulsed, `robot-movement-error*` is set to the error code (if any), and `robot-angle-diff*` is set to the estimated amount turned.

`rpl-robot-go-forward` *time*
`rpl-robot-goto-wall`
`rpl-robot-follow-wall`
`rpl-robot-goto-designator` *cam*
These commands move the robot, pulsing `robot-stopped*` when done and setting `robot-movement-error*`; `rpl-robot-goto-designator` also sets `robot-arrived-at*` to the designator arrived at (if any).

`rpl-robot-stop`
Stop the robot and pulse `robot-stopped*`.

`rpl-robot-place`

Sense the robot's current place type and put it into `robot-place*`. Pulses `robot-place-input*` when done.

Camera fluents

Keeping track of camera status is done through a set of fluent vectors, indexed by camera number. So, `(vref` *fluent-vec i*`)` gives the fluent for `(robot-camera` *robot i*`)`. The fluent vectors are as follows:

`camera-input*` Pulsed when new sensory information is available for a camera.

`camera-views*` Contains the latest view computed for each camera.

`camera-designators*` Contains a list of designators from the last designator sensing operation for each camera.

`camera-error*` The last sensing error code for each camera.

`camera-tracking*` Pulsed when a camera starts tracking a designator.

`camera-tracking-error*` The code for each camera's last tracking error.

`camera-moved*` Pulsed when a camera completes a move.

`camera-movement-error*` The code for each camera's last movement error.

Camera functions

`rpl-camera-field-of-view` *c*
`rpl-camera-resolution` *c*

These accessors return the 'optical' parameters of camera *c*.

`rpl-camera-angle` *c*

Returns the relative angle of camera *c* with respect to the robot.

`rpl-camera-last-view` *c*

Returns the last view sensed by camera *c*.

`rpl-camera-turn` *c ang*
`rpl-camera-align-place` *c*
`rpl-camera-align-robot` *c*

Turn camera *c*, pulsing `(vref camera-moved*` *c*`)` when completed and setting `(vref camera-movement-error*` *c*`)` to the error code returned.

`rpl-camera-view` *c*

Compute the current view for camera *c* and put it into `camera-views*`. Pulse `(vref camera-input*` *c*`)` when done.

`rpl-camera-get-object-designator` *c type specs*
`rpl-camera-get-all-object-designators` *c type specs*
`rpl-camera-get-place-designator` *c type*
`rpl-camera-get-all-place-designators` *c type*
Use camera *c* to find designators of various kinds, pulsing `camera-input*` when done and setting `camera-error*` to any error code produced. A list of the designators found is placed in `camera-designators*`.

`rpl-camera-track` *c desig*
`rpl-camera-untrack` *c*
Start or stop a camera tracking. When tracking starts, `camera-tracking*` is pulsed; if an error occurs, `camera-tracking-error*` is set to the error code.

`rpl-camera-insert` *c desig*
`rpl-camera-remove` *c*
Insert or remove the camera into or out of an object, pulsing `camera-moved*` when done and putting error codes into `camera-movement-error*`.

Hand fluents

Hand status is represented by fluent vectors in the same way as camera status.

`hand-moved*` Pulsed whenever a hand completes a movement.

`hand-error*` Set to the last error code signalled for each hand.

`hand-lost*` Pulsed when a hand loses its grasp.

`hand-inside*` Set to the designator or bin a hand is inside.

`hand-grasping*` Set to the designator or bin a hand is grasping.

`hand-input*` Pulsed when sensory information becomes available for a hand.

Hand functions

`rpl-hand-strength` *h*
Returns the strength of robot hand *h*.

`rpl-hand-insert` *h desig*
`rpl-hand-insert-bin` *h bin*
`rpl-hand-remove` *h*

Move the hand into and out of objects or storage bins, pulsing `hand-moved*` when done and setting `hand-error*` to the resultant error code. When an insertion is successful, `hand-inside*` is set to the object of the insertion (a designator or bin).

`rpl-hand-grasp-desig` *h c desig*
`rpl-hand-ungrasp` *h*

Grasp or ungrasp an object; pulse `hand-moved*` when done and set `hand-error*` If a grasp is successful, `hand-grasping*` is set to the object of the grasp (a designator).

`rpl-hand-grasp-check` *h*
`rpl-hand-track` *h*

Check or track grasping status, respectively, setting `hand-error*` to the error code produced. When `rpl-hand-grasp-check` completes, it pulses `hand-input*`, whereas `hand-lost*` is pulsed when `rpl-hand-track*` detects a lost grasp.

Storage bin fluents

Storage bin fluents also come in vectors, as above.

`storage-bin-moved*` Pulsed when a bin movement (open/close) completes.

`storage-bin-error*` Set to the last error code for a bin.

`storage-bin-jammer*` If a bin cover is jammed by a robot part, the corresponding fluent here is set to the jamming part.

Storage bin functions

`rpl-storage-bin-capacity` *b*
`rpl-storage-bin-real-capacity` *b*

Return a storage bin's total capacity and current capacity, respectively.

`rpl-storage-bin-closed?` *b*

Return true if storage bin *b* is closed, otherwise false.

`rpl-storage-bin-open` *b*
`rpl-storage-bin-close` *b*

Open or close a storage bin, pulsing `storage-bin-moved*` when done and setting `storage-bin-error*` to the error code produced. If the bin jams on closing, the offending robot part is put into `storage-bin-jammer*`.

Resource functions

`rpl-grab-robot`
`rpl-grab-camera` *c*
`rpl-grab-hand` *h*
`rpl-grab-bin` *b*

These functions grab the specified resources, essentially stopping other processes running on them.

Appendix D

An Extended Example

This appendix presents a detailed example of a robot program to aid in understanding how to use the simulator. Our objective is to write a routine that will find an object with a given description (nearby), go to it, grasp it, carry it in a robot storage bin back to its starting point, and drop it on the ground.

Scanning for an object

We start with a routine that will find an object with a given descriptions, returning a designator for further use. In Figure D.1 we define a function that continually turns a camera and looks for a matching object. This routine turns the camera to face an unseen area of the world. When the turn is completed, we either abort if an error occurred, or try to get a designator for the described object. If no designator is found, we repeat until one is. Note that this function that we've defined takes a continuation in the same way as built-in robot commands, since it returns immediately, after scheduling the proper events in the robot's 'controller'. Figure D.2 shows a RPL version of the same function. This version is somewhat clearer, due to the use of iteration and explicit synchronization. Also note the use of a fluent parameter for returning the designator found.

Going to an object

Once we can scan for an object from one position, we now write a routine (see Figure D.3) to move in a straight line while scanning, which returns the designator found (or an error if none). First we fire off two control processes, one for scanning as discussed above, and one for moving forward a specified distance. The routine then waits until something happens, and either goes to the designator found, or returns an error. This shows the use of continuations for setting status variables in order to synchronize the robot and the world. If you use RPL, however (Figure D.4), the necessary synchronization can be done using parallelism and policies. The RPL procedure also returns the enhanced object descriptor for use later. Another fact to note is the necessity of

```
(deffunc scan-for-object - VOID (cam - CAMERA type - SYMBOL spec - THING-DESCRIPTOR
                                 cont - ROBOT-CONTINUATION)
  (camera-turn cam (!_field-of-view cam)
               (continuation (err cam ang)
                 (if err
                     (funcall cont err nil)
                     (camera-get-object-designator
                      cam type
                      spec (continuation (err cam desig)
                             (cond ((null desig)
                                    (scan-for-object cam spec cont))
                                   (t
                                    (funcall cont nil desig)))))))))
```

Figure D.1: Continuation-based object scanning function.

```
(def-interp-proc rpl-scan-for-object (cam type spec desig)
  (let ((got-desig (state 'got-desig)))
    (loop UNTIL got-desig
      (rpl-camera-turn cam (rpl-camera-field-of-view cam))
      (wait-for (vref camera-moved* cam))
      (if (not (null (fluent-value (vref camera-movement-error* cam))))
          (fail)
          (seq
            (rpl-camera-get-object-designator cam type spec)
            (wait-for (vref camera-input* cam))
            (if (vref camera-designators* cam)
              (seq
                (set-value got-desig T)
                (set-value desig
                           (car (fluent-value (vref camera-designators* cam)))))))))
    got-desig))
```

Figure D.2: RPL object scanning function.

```
(deffunc go-to-object - VOID (rob - ROBOT cam - CAMERA spec - THING-DESCRIPTOR
                              timeout - INTEGER cont - ROBOT-CONTINUATION)
  (let ((desig NIL) - (~ DESIGNATOR)
        (stop #F) - BOOLEAN)
    (scan-for-object cam :NON-LOCAL spec (continuation (err des)
                                            (unless err
                                              (!= desig des))))
    (robot-go-forward rob timeout (continuation (err rob)
                                    (!= stop #T)))
    (unwind-protect
        (loop UNTIL (or desig stop) (nop))
      (if desig
          (progn (camera-track cam desig (continuation (err &rest fu) ()))
                 (robot-goto-designator rob cam (continuation (err &rest fu)
                                                  (grab-resource cam)
                                                  (funcall cont err desig))))
          (funcall cont :NO-OBJECT-FOUND rob cam NIL)))))
```

Figure D.3: Go to designator function.

```
(def-interp-proc rpl-go-to-object (cam spec timeout)
  (rpl-robot-set-speed .3)
  (let ((desig (create-fluent 'desig NIL)))
    (pursue (rpl-scan-for-object cam :NON-LOCAL spec desig)
      (seq (rpl-robot-go-forward timeout)
           (wait-for robot-stopped*)))
    (if desig
        (with-policy (seq (set-value (vref camera-tracking-error* cam) nil)
                          (whenever (vref camera-tracking-error* cam)
                            (wait-time 1)
                            (if (not robot-arrived-at*)
                                (seq
                                 (print (vref camera-tracking-error* cam))
                                 (fail )))))
          (seq (rpl-camera-track cam desig)
               (wait-for (vref camera-tracking* cam))
               (set-value robot-stopped* nil)
               (set-value robot-arrived-at* nil)
               (rpl-robot-goto-designator cam)
               (wait-for (and robot-stopped* robot-arrived-at*))))
        (fail))
    (rpl-designator-data desig)))
```

Figure D.4: Go to designator RPL procedure.

manually resetting fluents such as `camera-tracking-error*` to known states before relying on their values. In a more complex plan, valves would probably be used to guarantee ownership of various resources.

Getting the object

The next stage is to actually get the target object. Figures D.5 and D.6 show procedures to accomplish this. The sequence of events (somewhat clearer in the RPL code) is:

1. Get a local designator and track it
2. Grasp the object
3. Open the storage bin and insert the hand into it
4. Ungrasp the hand (dropping the object)
5. Remove the hand from the bin and close the bin.

This plan exhibits more complex robot coordination than the previous ones. The plan still is not terribly robust—it simply fails when errors occur. Making the plan more robust by using error-recovery strategies is left as an exercise for the reader.

Putting the object down

The plan for putting an object down (that is, taking it out of a storage bin and dropping it) is similar to that for grabbing it. The one significant difference is the necessary insertion of a camera into the storage bin, to sense the target object for grasping. We only show the RPL procedure (in Figure D.7) for clarity; the continuation-passing version is analogous.

Returning home

Returning to home base based purely on odometric information is bound to be inaccurate. But it is the simplest thing to do, so we shall assume that it will work well enough. The basic idea is to start an odometer counting before executing some procedure, and then after the procedure finishes, to maneuver the robot such that the center of the odometric interval (the best estimate of relative position) is near the origin. A general RPL macro to do this is shown in Figure D.8. A new odometer is created, the desired body is executed, and then the robot repeatedly turns toward the origin and moves forward. This will clearly fail if there are obstacles; again, robustness is left as an exercise for the reader.

```
(deffunc grab-object - VOID (h - HAND cam - CAMERA spec - THING-DESCRIPTOR
                             bin - STORAGE-BIN cont - ROBOT-CONTINUATION)
  (scan-for-object cam :LOCAL spec
        (continuation (err cam desig)
           (if (null desig)
               (funcall cont :NO-DESIGNATOR-FOUND nil))
               (camera-track cam desig (continuation (err &rest fu) ()))
               (hand-grasp-desig h cam desig
                 (continuation (err &rest fu)
                   (if err
                       (funcall cont err desig))
                       (storage-bin-open bin
                          (continuation (err &rest fu)
                             (hand-insert-bin h bin
                                (continuation (err &rest fu)
                                   (hand-ungrasp h
                                    (continuation (err &rest fu)
                                       (funcall cont err desig)))))))))))
```

Figure D.5: Object grabbing function.

```
(def-interp-proc rpl-grab-object (h cam spec bin)
  (set-value (vref camera-designators* cam) '())
  (set-value (vref camera-error* cam) nil)
  (let ((desig (create-fluent 'desig nil)))
    (rpl-scan-for-object cam :LOCAL spec desig)
    (if (vref camera-error* cam)
        (fail))
    (set-value (vref camera-tracking-error* cam) nil)
    (with-policy (seq (set-value (vref camera-tracking-error* cam) nil)
                      (whenever (vref camera-tracking-error* cam) (fail)))
      (rpl-camera-track cam desig)
      (wait-for (vref camera-tracking* cam)))
    (with-policy (seq (set-value (vref hand-error* h) nil)
                      (whenever (vref hand-error* h) (fail)))
      (with-policy (whenever (vref camera-tracking-error* cam) (fail))
        (rpl-hand-grasp-desig h cam desig)
        (wait-for (vref hand-moved* h))
        (rpl-storage-bin-open bin)
        (wait-for (vref storage-bin-moved* bin))
        (rpl-hand-insert-bin h bin)
      (wait-for (vref hand-moved* h)))
      (rpl-hand-ungrasp h)
      (wait-for (vref hand-moved* h))
      (rpl-hand-remove h)
      (wait-for (vref hand-moved* h))
      (rpl-storage-bin-close bin)
      (wait-for (vref storage-bin-moved* bin)))))
```

Figure D.6: Object grabbing RPL procedure.

```
(def-interp-proc rpl-put-down-object (h cam spec bin)
  (rpl-storage-bin-open bin)
  (wait-for (vref storage-bin-moved* bin))
  (with-policy (seq (set-value (vref camera-movement-error* cam) nil)
                    (whenever (vref camera-movement-error* cam) (fail)))
    (rpl-camera-insert cam bin)
    (wait-for (vref camera-moved* cam)))
  (let ((desig (create-fluent 'desig nil)))
    (with-policy (seq (set-value (vref camera-designators* cam) '())
                      (set-value (vref camera-error* cam) nil)
                      (whenever (vref camera-error* cam) (fail)))
      (rpl-scan-for-object cam :LOCAL spec desig)
      (set-value (vref camera-tracking-error* cam) nil))
    (with-policy (seq (set-value (vref camera-tracking-error* cam) nil)
                      (whenever (vref camera-tracking-error* cam) (fail)))
      (rpl-camera-track cam desig)
      (wait-for (vref camera-tracking* cam)))
    (with-policy (seq (set-value (vref hand-error* h) nil)
                      (whenever (vref hand-error* h) (fail)))
      (rpl-hand-insert-bin h bin)
      (wait-for (vref hand-moved* h))
      (with-policy (whenever (vref camera-tracking-error* cam) (fail))
        (rpl-hand-grasp-desig h cam desig)
        (wait-for (vref hand-moved* h)))
      (par
       (seq (rpl-hand-remove h)
            (wait-for (vref hand-moved* h))
            (rpl-hand-ungrasp h)
            (wait-for (vref hand-moved* h)))
       (seq
        (rpl-camera-remove cam)
        (wait-for (vref camera-moved* cam))
        (rpl-camera-untrack cam)))))))
```

Figure D.7: RPL procedure for putting an object down.

```
(def-interp-macro (with-odometric-return ?name . ?body)
  `(let ((odo (rpl-new-odometer ',name)))
     ,@body
     (let ((nom (nominal (ars:get-odometer-pos odo))) - PT)
       (with-policy (seq (set-value robot-movement-error* nil)
                         (whenever robot-movement-error* (fail)))
         (loop UNTIL (< (dist nom origin*) .5)
           (let ((ang (nominal (ars:get-odometer-ang odo))))
             (rpl-robot-turn (- (atan2 (- (!_(PT y) nom)) (- (!_(PT x) nom)))
                                ang))
             (wait-for robot-stopped*)
             (rpl-robot-go-forward 1)
             (wait-for robot-stopped*))
           (!= nom (nominal (ars:get-odometer-pos odo))))))))
```

Figure D.8: RPL macro for returning to (approximately) a starting position.

Putting it all together

The final top-level RPL plan is given in Figure D.9. The robot goes and gets the object, returns, and deposits it on the ground. Simplicity itself, at this level.

```
(def-interp-proc get-object-and-return (h cam spec bin walk-time)
  (seq (with-odometric-return temp-odom
         (rpl-go-to-object cam spec walk-time)
         (rpl-grab-object h cam spec bin))
       (rpl-put-down-object h cam spec bin)))
```

Figure D.9: Top-level RPL procedure for getting an object and bringing it back.

Appendix E

The Command-Line Interface

The Command Line Interface (CLI) is an interactive user interface for controlling the robot. Note that the CLI takes the place of a robot program, so only primitive robot operations are provided, which may seem cumbersome at times. The prompt `Robot>` signifies that the CLI is running. Commands to the CLI consist of a command name followed by a set of arguments separated by spaces. Command lines delimited by parens are evaluated as Lisp expressions, providing a way to escape to Lisp. All cameras, hands, designators and storage-bins are specified by their numeric index in the CLI. The indices are also used for display.

CLI Messages

All robot operations are accompanied by messages, which are displayed after an operation is finished. Since most actions take some time to complete, messages may arrive at any time. Messages come in three forms: messages, warnings and errors (as discussed in Section 3.7) . Whenever an operation completes successfully, a message is returned stating the result of the operation. Warnings, preceeded by `*WARNING* ->`, inform the user that an event outside the robot's control has occurred as a consequence of its action. For example, warnings are displayed when designators are lost due to the robot moving. Errors, preceeded by `*ERROR* ->`, are issued when the robot fails in a task.

Built-in Commands

Built-in commands basically serve the purpose of aiding the user in performing tests on the simulator and determining the status of the robot at certain times. The following built-in commands are defined in the CLI:

`ALIAS [Alias Cmd]`: Displays the currently defined aliases, if no arguments are present, or allows for the definition of a new alias for `Cmd`.

`EVAL Exp`: Evaluates `Exp` as a LISP expression.

`REPEAT Num Cmd`: Repeats the command `Cmd` a `Num` number of times.

`HELP [Cmd]`: Displays the main command menu, if no argument is present, otherwise help is displayed for the specified command. The main menu shows all of the built-in commands as well as those that affect the robot's movement and turning.

`QUIT`: Exits the CLI.

`MENU-BIN`: Displays the menu of possible commands for manipulating storage-bins.

`MENU-CAMERA`: Displays the menu of possible commands for manipulating cameras.

`MENU-HAND`: Displays the menu of possible commands for manipulating hands.

`BINS`: Displays the various storage-bins on the robot, including their status (`open` or `closed`), total capacity, current capacity and contents.

`CAMERAS`: Displays the status of the cameras on the robot, including field of view, resolution, current location and designator being tracked, if any.

`HANDS`: Displays all of the hands on the robot, including their strength, location and what they are grasping.

`THINGS`: Displays a list of all the objects in the world together with their locations and attributes.

`STATUS`: Displays the status of the robot, including position, last change in position, last estimated change in position, current heading, speed, turning speed, the views from all of its cameras and any currently valid designators.

There are two more built-in commands, `POS x y` and `ANGLE ang`. These commands instantaneously set the position and angle of the robot respectively, without any error (and ignoring obstacles).

Robot Commands

For most robot control functions discussed above, there is a corresponding command in the CLI. These are fairly self-explanatory with a few exceptions. First, all cameras, hands, storage bins, and designators are specified with identification numbers instead of the actual structures. Secondly, the robot is never specified, it is always assumed[1]. Finally, some commands differ slightly in other ways from the function they roughly correspond to. The following list presents these differences:[2]

`OBJ-DESIG cam type specs`: Attempts to find an object designator using the specified camera, where `type` is either `local` or `non-local` and `specs` is the list of attributes to match the object against.

[1]Note that when multiple agent capability is added, this assumption will have to be changed.

[2]Remember that the arguments `cam`, `hand`, `bin` and `desig` refer to the identification numbers of the respective objects.

PLACE-DESIG cam place: Attempts to find a place designator. Note that the type does not have to be specified since it is assumed to be non-local.

CAM-INSERT-DESIG cam: Inserts the specified camera into the local designator which it is tracking.

CAM-INSERT-BIN cam bin: Inserts the camera into the specified storage-bin.

HAND-INSERT-BIN hand bin: Inserts the specfied hand into specified storage-bin.

HAND-INSERT-DESIG hand cam: Inserts the hand into the local object designator tracked by cam.

A CLI Example

Following is an example session with the CLI. Lines in sans serif are typed by the user.

```
Robot> obj-desig 0 local ((:size 5))
Robot> get-designator: Camera 0 found Designator 0
Robot> cam-track 0 0
Robot> camera-track: Camera 0 tracking Designator 0
Robot> cam-track 1 0
Robot> camera-track: Camera 1 tracking Designator 0
Robot> cameras
CAMERA FIELD RES ANGLE
------ ----- --- -----
   0     0.79  3  -5.75 Tracking Local Object designator @ 0.00, -5.75
   1     1.57  5  -5.75 Tracking Local Object designator @ 0.00, -5.75
Robot> cam-insert-desig 0
Robot>
  *WARNING* -> Camera 0 moved. No longer tracking Designator 0
camera-insert: Camera 0 inserted into SMALL-BOX#3
Robot> obj-desig 0 local ((:size 1 2))
Robot> get-designator: Camera 0 found Designator 1
Robot> cam-track 0 1
Robot> camera-track: Camera 0 tracking Designator 1
Robot> status
  Position:                (20.00,20.00)
  D Position:              (0.00,0.00)
  Angle:                   -2.65
  Speed:                   8
  Turn Angle:              0.30
  View:  Camera 0: TRACK 121.94 121.56 119.71
         Camera 1: TRACK 79.85 76.62 11.80 10.40 14.80
  Place:                   NIL
  Designators:
   0  Local Object @ 0.00, -5.75
      :SUBSTANCE Wood  :CAPACITY 6.4  :OBSMOVE No  :OBSVIEW No
      :SIZE 5.1
   1  Local Object @ 0.00, -5.73
      :OBSVIEW No  :SIZE 0.2
Robot>  hand-grasp 0 0
Robot>
  *ERROR* -> Hand 0 not in same location as Camera 0
Robot>  hand-insert-desig 0 1
Robot> hand-insert: Hand 0 inserted into SMALL-BOX#3
Robot>  hand-grasp 0 0
Robot> hand-grasp: Hand 0 grasped PEN#2
Robot>  hand-remove 0
Robot>
  *WARNING* -> PEN#2 has moved
  *WARNING* -> Removing Designator 1
  *WARNING* -> Designator 1 no longer valid. Camera 0 lost track
hand-remove: Hand 0 removed from SMALL-BOX#3
Robot>  hand-insert-bin 0 1
Robot>
  *WARNING* -> PEN#2 has moved
hand-insert: Hand 0 inserted into Storage-bin 1
Robot>  hands
HAND STRENGTH GRASPING
---- -------- --------
  0      10    PEN#2          Located in storage-bin.
  1       4
Robot>  hand-ungrasp 0
Robot> hand-ungrasp: Hand 0 dropped PEN#2 into Storage-bin 1
Robot>  bin-close 1
Robot>
  *ERROR* -> Hand 0 still in Storage-bin 1
Robot>  hand-remove 0
Robot> hand-remove: Hand 0 removed from Storage-bin 1
Robot>  bin-close 1
Robot> storage-bin-close: Storage-bin 1 closed
Robot>  bins
BIN STATUS TOTAL-CAP CURRENT-CAP CONTENTS
--- ------ --------- ----------- --------
 0   Open      20         20.0
 1  Closed      5          4.8    PEN#2
```

Here the robot is looking for an object inside of another object. The first command finds a local object designator using camera 0. Note that both cameras are set to track the designator. This is necessary since one camera is going to move inside the object and we want the designator to remain so that a hand can follow the camera into the object. The **cameras** command shows that the cameras are indeed tracking the same object. When camera 0 is inserted, it loses its track, but designator 0 remains, because it is being tracked by camera 1. Camera 0 now locates another object inside of SMALL-BOX#3. The **status** command shows what attributes the designated objects have. Note that the attribute lists are not necessarily accurate or complete. Also, the view of camera 0 consists basically of several samples of the same number since this is the substance value of SMALL-BOX#3. In order for hand 0 to grasp PEN#2, it has to be in the same location as camera 0. PEN#2 is grasped and removed and then inserted into storage-bin 1, which can not close until PEN#2 is released and hand 0 has exited the storage-bin. Each time a hand moves with an object, a warning is issued in order to explain the possible disappearance of designators.

Appendix F

Index

Commands

Functions and Macros

Parameters

Data Types

RPL Interface

RPL Fluents

Error Codes