Minimalism + Distribution = Supermodularity

Bruce Donald
Department of Computer Science
Cornell University
Ithaca, NY 14853

Jim Jennings
Department of Computer Science
Tulane University
New Orleans, LA 70118

Daniela Rus
Department of Computer Science
Dartmouth College
Hanover, NH 03755

April 18, 1996

Abstract

We have designed and implemented multi-agent strategies for manipulation tasks by distributing mechanically-based sequential algorithms across several autonomous spatially-separated agents, such as mobile robots. Our experience using mobile robots for the manipulation of large objects (couches, boxes, file cabinets, etc.) leads us to recommend a minimalist architecture for multi-agent programming. In particular, our methodology has led us to derive asynchronous distributed strategies that require no direct communication between agents, and very sparse geometric and dynamic models of the objects our robots manipulate.

We argue for a design principle called supermodularity*, which is orthogonal both to the notion of modularity in cognitive AI and also to horizontal decomposition (the non-modularity advocated in the subsumption/connectionist literature.)*

Finally, we discuss a simple MOBOT-SCHEME *infrastructure to implement supermodular architectures. In the past few years we have programmed many supermodular manipulation protocols and tested them extensively on our team of mobile robots. We describe why we think the supermodular infrastructure results in robust, simple, readable, manipulation strategies that can be recycled and reused.*

1 Introduction

In robotics most manipulation algorithms are designed to execute in a single process on a single computer that takes input from all the sensors and controls all the effectors. To develop distributed manipulation strategies, we began with a sequential but mechanically-based robot algorithm for pushing and grasping (*e.g.*, [Mas95]). While quite general in principle, these off-line algorithms are usually designed for robotics devices such as grippers or fingers attached to a traditional robot arm. To extend these results for distributed manipulation (defined in this paper in Section 5) there are several challenges. (1) Autonomous mobile robots (mobots) have a suite of sensing and control modalities that differ across robot architectures. (2) Mobots are often better suited to on-line approaches and hence the algorithms must be adapted to rely less extensively on geometrical and dynamical models. (3) A host of difficulties arises when a task must be performed by a distributed team instead of a single agent—and hence the algorithms must compensate by changing their communication, sensing, or knowledge requirements[Don95, DJR93, DJR94a, RDJ95].

We have reported on several sets of manipulation strategies (which we call *protocols*), and also on the methodology which generated them [Don95, BBD$^+$95, RDJ95, Jen95]. Our most interesting protocols are asynchronous and do not require communication between the agents. In this paper we address the architecture and the programming environment we used to develop our protocols. We also discuss how the class of distributed manipulation algorithms we have developed leads naturally to certain architectural constraints.

This paper is organized as follows. We begin with a discussion on minimalism and supermodularity. We continue by describing three different minimalist manipulation protocols we have implemented and analyzed in the information invariants framework, and we highlight their supermodular structure. Then we describe the infrastructure we used for developing our strategies. Finally, we discuss how supermodularity and minimalism affect distributed robot architectures.

2 Minimalism

Minimalism pursues the following agenda: For a given robotics task, find the minimal configuration of resources required to solve the task. Thus, minimalism attempts to reduce the resource signature for a task, in the same way that (say) Stealth technology decreases the radar signature of an aircraft. Minimalism is interesting because doing task A without resource B proves that B is somehow inessential to the information structure of the task. We will discuss our experimental demonstrations and show how their implementation relates to our theoretical proofs of minimalist systems. In particular, we will describe our MOBOT SCHEME system—a distributed, multi-threaded, high-level robot programming environment.

In robotics, minimalism has become increasingly influential. Marc Raibert showed that walking and running machines could be built without static stability. Erdmann and Mason showed how to do dextrous manipulation without sensing. Tad McGeer built a biped, kneed walker without sensors, computers, or actuators. Rod Brooks has developed online algorithms that rely less extensively on planning and world-models. Canny and Goldberg have demonstrated robot systems of minimal complexity. We have taken a minimalist approach to distributed manipulation (defined in this paper in Section 5) and also to our choice of a software architecture for writing our robot programs. We claim that the resulting protocols consume a near-minimum amount of resources. The accompanying software development, debugging, and execution-time system are concomitantly parsimonious and lean.

3 Supermodularity

This paper introduces the concept of supermodularity, and in particular, it brings to the foreground the supermodularity of our manipulation strategies. In programming, it is natural to talk about units of organization called subroutines. In mobile robotics—particularly for distributed strategies—we have developed analysis and synthesis tools for units of organization called *circuits* [Don95]. Intuitively, a circuit is a sensori-computational unit consisting of sensors and actuators connected by data paths. We model our circuits as graphs. Vertices correspond to different sensori-computational components. Edges correspond to the data paths through which the information passes. Circuits can be transformed by changing the edge or vertex structure of their graphs. Different immersions of the graphs correspond to different spatial allocations of resources. An important class of transformations consists of permutations. A circuit permutation is a vertex permutation followed by an edge permutation of its graph.

Roughly speaking a subroutine is *modular* if it can be reused without changing the interface. We say a circuit is *supermodular* if it can be relocated to a different physical location and still function correctly even in the absence of circuits that formerly surrounded it and in the presence of new circuits at this new location. In this paper we will describe supermodular circuits according to the following hierarchy of circuit transformations. A circuit is *replicated* when it is duplicated at the same, or at a different location. A circuit is *distributed* when it is split up and the parts are recombined to form different circuits. A *supermodular permutation* consists of moving circuits around while respecting supermodular boundaries.

The chief debate between cognitive AI and architecturally constrained approaches, such as subsumptionism and connectionism, is as follows: cognitive AI advocates that skills should be modular. Connectionism/subsumption enforces a particular architecture above all else and this architecture may violate modularity. We maintain that this so-called dichotomy opposes the wrong categories: the issue is not modularity vs. non-modularity—this faux dichotomy arises from only considering single-agent systems in which all resources are physically co-located—in this case supermodularity reduces (or more accurately, masquerades) as simple, naked modularity.

On the other hand, in distributed systems, resources are of course not physically co-located.

In order to achieve the following goals:

- simplicity
- ease of reuse
- performance guarantees
- fault tolerance

we choose the design constraint of supermodularity. We will show how supermodularity achieves this goal. Since this design constraint defines an architecture, our architecture could also be called "supermodularity".

As an example of a supermodular circuit illustrating these points, we discuss in detail two circuits called `(align)` and `(push-track)` (see Section 5.2). `(align)` is a virtual orientation sensor which by actively exerting compliant control changes the robot's heading to lie at a particular, desired angular relation to a manipulated object. `(push-track)` is a virtual effector that given a velocity control, executes a guarded move while in contact, to apply a force along a desired line-of-pushing to a manipulated object. We will describe a supermodular protocol called ASYNC-ONLINE that uses `(push-track)` for multi-robot reorienting of large objects.

Supermodularity is a continuum. A circuit X is said to be *more supermodular* than circuit Y if the region C(X) of the configuration space $\mathcal{C}$ in which X functions correctly strictly contains C(Y). A *completely supermodular* circuit Z has C(Z) = $\mathcal{C}$. Hence, supermodularity is a partial order on circuits. In this paper we will (informally) use the term "circuit X is *often supermodular*" to connote that "C(X) is large". This concept may be quantified precisely by measuring the relative volume of C(X) in $\mathcal{C}$ [DRJ96]. In this paper we describe a partial order on the circuits we describe in detail. Some circuits are completely supermodular for the manipulation tasks we consider, while other circuits are not supermodular at all. We will show that ASYNC-ONLINE is completely supermodular, and more supermodular than `(push-track)`, which in turn is more supermodular than `(prim-push)`, the circuit for applying a force along a pre-specified line of pushing.

Our mantra for this paper is as follows:

"Minimalism + Distribution = Supermodularity"

In other words, simplicity does not arise from supermodularity but rather, from minimalism. Similarly, the transformations we permit circuits to undergo are constrained by the information invariants theory. That is, minimalism defines what it means for circuits to be simple and information invariants define what it means for circuits to be distributed. When the two are combined, the "optimal" kind of circuits are the supermodular ones.

Supermodularity yields recyclable and portable code. The greatest challenge is to formulate a method for authoring optimal supermodular circuits with *performance guarantees.* Our research agenda for performance guarantees is (i) to ensure performance guarantees for individual circuits; (ii) to show that a supermodular circuit with performance guarantees retains these guarantees when relocated; and (iii) to show that a supermodular circuit with performance guarantees retains these guarantees when distributed and parallelized. More specifically, (i) ensures that a circuit has a predictable functionality; for example it performs to within specified accuracy as in the case of the `(align)` circuit discussed in Section 5.2. In addition, (ii) ensures that the circuit has the same functionality when moved to a different location. The goal here is to show that under any permutation that respects co-designation constraints ([Don95]), the supermodular circuits retain, at least locally, their performance guarantees. Finally, (iii) ensures that the performance guarantees of a supermodular circuit are preserved in the presence of other circuits or agents that might (*a priori*) interfere with its basic functionality.

We have already derived performance guarantees for sequential and single agent circuits. An example is `(align)` whose analysis is described in [JR93] (see Section 5.2). Arguments that rely on geometry and task mechanics can also be made about our other supermodular circuits. We believe that because our manipulation strategies are performed in quasi-static environments (Section 6.1), given velocity bounds, the performance guarantees of single agent sequential circuits will be preserved when distributed. We tried to demonstrate this point empirically by running a large number of tests and experiments. While these results are yet to be proved rigorously, we believe that the supermodular framework is the right architecture for addressing performance guarantees.

4 Previous Work

There has been much work on cooperative systems of robots. For an excellent review see [CFKM95]. Previous work on cooperative manipulation has focused mostly on pushing, usually in the context of box-pushing by multiple mobile robots, *e.g.*, [Par94], [Nor93].

4.1 Robotic Manipulation

[MS85] presents extensive analysis of grasping and pushing operations under a quasi-static model. Mason's analyses of the mechanics of pushing and grasping have led to many practical manipulation strategies implemented most often on anthropomorphic robot arms with simple two-finger grippers. Similarly, [Bro85] and others have analyzed the geometry and mechanics of quasi-static pushing and squeeze-grasping of planar objects with common "parallel jaw" grippers.

Some work has been done on large-scale manipulation using a single mobile robot, such as [LM94] and [OY92]. The former analyze the mechanics of planar pushing with line contact (*e.g.*, a mobile robot with a fixed flat blade pushing a box) and demonstrate a manipulation planner which maintains this contact configuration.

4.2 Cooperating Mobile Robots

Other recent work investigates tasks in which multiple mobile robots cooperate, such as: [SB93] (manipulation of pallets by many small robots in simulation); [Mat93] (study of group behaviors such as dispersion and flocking); [ABN93] (simulation of foraging agents with and without communication).

Work combining cooperation with mobot manipulation includes [Nor93], [Par94], and [DJR93, DJR94a, DJR94b, RDJ95, Jen95]. Each demonstrates the manipulation of a box or other large object using two mobile robots. [Nor93] describes a task in which one robot pushes a box and another robot clears obstacles out of the way. [Par94] builds an architecture designed to achieve fault-tolerant cooperation within teams of heterogeneous mobile robots and applies that architecture to a number of tasks (mostly in simulation), including "hazardous waste cleanup," in which several mobots cooperate to pick up a number of small objects and move them near a designated site.

One task from [Par94] implemented on real (physical) robots is a two-robot, box-pushing task. Each robot pushes its own end of the box by some fixed amount, then waits for the other robot to push its end. This strategy can be executed by a single robot in the event of failure of the other robot; the key feature of the strategy, and indeed of Parker's architecture, is its fault-tolerance.

We note that although much work has been done in simulation of cooperative manipulation, there is little evidence that the results reflect the behavior of physical robots performing similar tasks. Often the mechanics and dynamics are poorly modeled, and unreasonable assumptions are made about available communication and sensing devices. However, Lynch and Mason ([LM94]) present a simulator and motion planner for single-robot pushing that is based in sound mechanics, and this simulator may yet be extended to multiple-robot manipulation. Also, [DJR93, DJR94a, DJR94c, DJR94b, Jen95] present algorithms for two-robot pushing and for multi-robot reorientation. They analyze their protocols with respect to usage of such resources as computation, retained state (*e.g.*, models), sensors, and communication. Much of this work has the theme of minimalism, asking such questions as "Can we design a manipulation strategy that requires no communication between the agents?"

In the next section, we summarize our most interesting implemented manipulation protocols, those for multi-robot pushing, reorientation, and pushing/steering manipulation of large objects. In the sections that follow, we discuss our focus on minimalism, our robot programming philosophy, and the nature of the distributed manipulation algorithms we have implemented.

5 Three Distributed Minimalist Manipulation Protocols

This section describes our experience in building minimalist distributed strategies for mobots that perform manipulation tasks. We describe the circuits that implement the protocols we developed for three manipulation tasks, and highlight their supermodularity. We differentiate between the properties of manipulation protocols by robots according to the following hierarchy. In a *manipulation task*, robots use forces to reorganize the robots' space. In a *parallel manipulation protocol*, two or more robots apply forces to the same coupled dynamical system. In a *distributed manipulation protocol*, the computation and control are distributed among the robots in a way that qualifies as a distributed computation.

We present three different tasks, the protocols we developed for these tasks, and two examples of supermodular circuits that we recycled and reused for each task protocol. The experiments provide empirical evidence for our belief that code authored in supermodular architectures is simple, predictable, parallelizable, and reusable.

Specifically, we describe protocols that allow a team of small autonomous mobile robots to cooperate to move large objects (such as couches). The robots run SPMD[1] and MPMD[1] manipulation protocols with no explicit communication. We developed these protocols by distributing off-line, sequential algorithms requiring geometric models and planning. The resulting parallel protocols are more on-line, have reduced dependence on *a priori* geometric models, require no communication, and are typically robust (resistant to uncertainty in control, sensing, and initial conditions).

We will discuss the circuits needed to implement the protocols. In particular, we will introduce the following circuits:

1. 2Mθ, the protocol for two-robot straight-line pushing,
2. `(prim-push)`, the circuit for applying a force along a specified line of pushing,
3. `(align)`, the circuit that allows a robot to position itself at a specified relative orientation with respect to a surface,

[1]SPMD (MPMD) = *<u>S</u>ingle (<u>M</u>ultiple) <u>P</u>rogram, <u>M</u>ultiple <u>D</u>ata.*

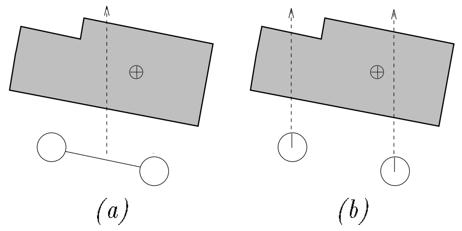


Figure 1: Two pushing tasks – the goal in each case is to push the block in in the direction indicated by the dashed arrows. *(a)* In the "two-finger" pushing task, as seen from above, the fingers (drawn as circles) are kinematically connected in the common "parallel-jaw gripper" configuration. *(b)* Although similar in appearance, the "two robot" pushing task is quite different. Each of the mobile robots (drawn as circles) is an autonomous machine, and there may be little or no explicit communication between them.

4. `(SPR)`, the circuit for a single pushing robot,

5. Async-online, the protocol for two robot reorientation,

6. `(push-track)`, the circuit for compliant pushing at an angle while sliding on the face,

7. Pusher-Steerer, the circuit for the MPMD manipulation system.

We will discuss how we used these circuits to implement the protocols for our three manipulation tasks, we will show how these circuits relate to each other, and we will discuss their supermodularity.

For all of our protocols, the manipulated objects have comparable size and dynamic complexity to the robots. Objects used in our experiments are up to six robot diameters in length, and up to twice the mass of one of the robots. Repositioning and reorientation of these objects may be possible only through active cooperation of a team of mobile robots. Employing multiple robots may yield performance benefits, or other advantages, such as ease of programming.

5.1 Pushing

Consider a task in which a single robot (manipulator) must push a box in a straight line. Figure 1a depicts this task, posed for a two-finger robotic manipulator. We do not assume that the robot possesses a complete model (geometric and physical) of the object it is pushing, or of the supporting surface. With a two-fingered push (see Figure 1a), the box will translate in a straight line so long as the COF (the center of friction) lies between the fingers. (The actual condition for stable straight-line pushing is slightly more complicated. See [MS85] for a complete analysis.) An advantage of the two-finger pushing strategy is that the COF can drift around some and yet the robot can keep pushing, since we only need ensure the COF lies in some region between the lines of pushing of the fingers (see Figure 1a), instead of on a line. If the COF moves outside the region, then the fingers can move sideways to "capture" it again. We have implemented a control loop for this task on our force-controlled Puma manipulator. The basic idea is to sense the reaction torque τ about the point $\oplus$ in Figure 1a. If $\tau = 0$, push forward in direction of the arrow. If $\tau < 0$ move the fingers to the right; else, move the fingers to the left.

We now derive a different version of this pushing strategy with a parallel jaw gripper for a system of two autonomous robots that can push an object. This new strategy relies on the observation that the information needed to determine the motion of the box is present in the angle θ between the normal to the face of the box n and the direction of pushing p. See Figure 4. We wish the new protocol to run on two autonomous mobile robots which will replace the fingers, as in Figure 1b. We can adapt the control loop to servo on θ instead of τ because our robots can use their pushbutton bumpers to measure the relative angle between their heading and the orientation of the face of the box [JR93]. Actually, the robots first measure θ_0 (the initial angle between n and p), and subsequently compare this value to the angle $\theta(t)$ measured at time t in order to infer the direction of motion of the box. A negative change in the value of this angle implies a clockwise rotation of the box. A positive change implies a counterclockwise rotation. The robots adjust their pushing location along the face of the box accordingly. This is an example of how the robots can use the task dynamics[2] to determine their next actions. Pseudo-code for this strategy (called Protocol 2Mθ) is shown in Figure 2 and the circuit for the protocol is shown in Figure 3.

This pushing protocol depends crucially on two resources for the mobots: the ability to sense the relative orientation of the object at the point of contact and the ability to apply a force along a specified line of pushing. The relative orientation sensor is implemented by using the circuit `(align)` described in

[2]An alternative to using task dynamics is to use explicit communication. The robots could exchange the sensed torque at their points of contact to determine the best contact point on the face of the object. A detailed description of this method is presented in [DJR93, DJR94a].

```
Left robot                       Right robot

θ₀ ← θ(0)                        θ₀ ← θ(0)
repeat:                          repeat:
  push {                           push {
        measure θ(t)                     measure θ(t)
        }                                }
        until θ(t) ≠ θ₀                  until θ(t) ≠ θ₀
  if θ(t) > θ₀                     if θ(t) > θ₀
        translate-left                   translate-left
  else translate-right             else translate-right
```

Figure 2: Protocol 2Mθ. A strategy for two autonomous robots to push an object in a straight line without communication.

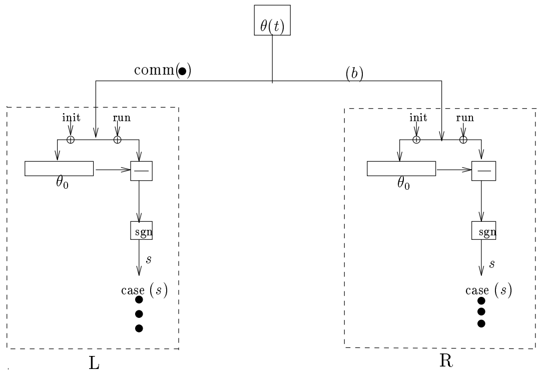


Figure 3: Sensor system for Protocol 2Mθ. This is a manipulation circuit for Protocol 2Mθ.

detail in Section 5.2 and Figure 8. Pushing is implemented by a circuit called `(prim-push)` (described in [RD92]).

5.1.1 Supermodularity in Pushing

`(prim-push)` gives us a simple version of pushing: it applies a force at a fixed direction with respect to the surface of the object. `(prim-push)` is a subcircuit that implements Protocol 2Mθ for cooperative straight-line pushing. In other words, `(prim-push)` $\subset$ 2Mθ.

A single robot system equipped with the circuit `(prim-push)` is denoted by `(SPR)`. `(SPR)` effects translations. This circuit is related to `(prim-push)` by the following information invariant equation:

$$(\texttt{SPR}) =_0 (\texttt{align}) + (\texttt{prim-push}), \qquad (1)$$

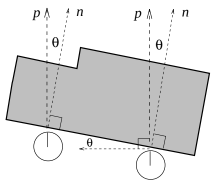


Figure 4: The critical quantities for the servo loop in Protocol 2Mθ.

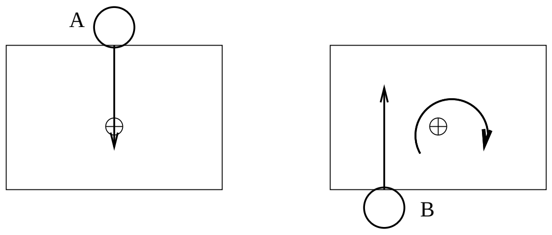


Figure 5: A single pushing robot is not a supermodular circuit. Relocating the circuit from A to B results in the application of the wrong torque sign.

where the composition of `(align)` and `(prim-push)` is parallel.

`(SPR)` is modular: it can be replicated on other robots for the same functionality. In our lab we have equipped three robots with similar but different architectures, TOMMY, LILY, and CAMEL[3], with `(prim-push)`.

`(prim-push)` is not a supermodular circuit, as its functionality (quantified by the applied torque) depends on its spatial location. When relocated, `(prim-push)` does not impart the same torque (see Figure 5). Moreover, a multiple robot system where each robot is `(SPR)` may effect translations, rotations, or the null operation (*i.e.*, the object stays in place), depending on their physical location. `(prim-push)` is our first example of a modular circuit that is not supermodular.

In contrast to `(prim-push)`, `(align)` is a supermodular circuit. The effect of align is to position a robot at a specified relative orientation with respect to a surface. `(align)` adapts to the surface normal to determine the heading of the robot no matter where the robot is located.

In the following section we describe circuits that are supermodular. Specifically, we show how

[3]CAMEL's differs mechanically from TOMMY and LILY in that it is a treaded rather than wheel-based robot and it contact bumpers are distributed on a line rather than a circular surface.

(prim-push) can be generalized to a circuit we call (push-track) so that it can apply a force at any desired orientation with respect to the surface of the object. This generalization is achieved by combining the circuits (prim-push) and (align) in a sequential, rather than parallel fashion. The sequential composition of (align) and (prim-push) results in a circuit sufficient for the reorientation task; the parallel composition of (align) and (prim-push) results in a circuit sufficient for the MPMD manipulation task. The details of the compositions will be given in Sections 5.2 and 5.3.

5.2 Reorientation

We are also interested in the reorientation of objects by teams of mobile robots. Consider the task whose goal is to change the orientation of a large object by a given amount. This is called the *reorientation* task. We have described and analyzed in detail the reorientation task in [RDJ95]. Figure 6 depicts one robot reorienting a large object. A robot can generate a rotation by applying a force that is displaced from the center of friction. This property relates the dynamics and the geometry of reorientations [Mas95] and it can be used to effect continuous reorientations with mobile robots. The idea is to compliantly apply a sliding force on the face of the object.[4] We call this action a *push-track* step.

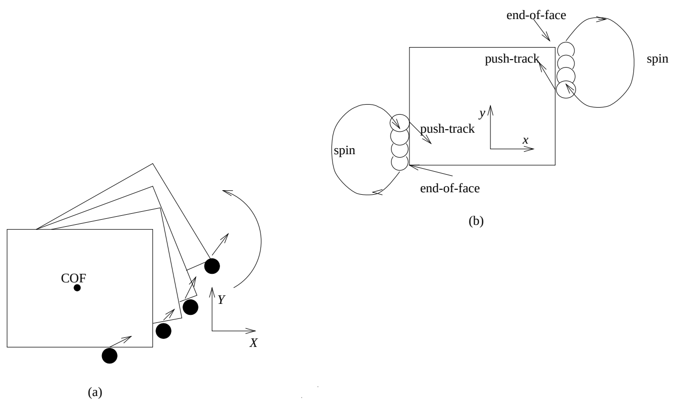
When the end of the face is reached, the robot may turn around to reacquire contact and repeat the push-tracking. A robot that has gone past the end of a face effectively losing contact with the object has *broken contact* with the object. A robot whose maximum applied force (defined by a threshold) does not change the pose of the object has encountered an *impediment*.

One robot may effect any desired reorientation by repeated push-tracking steps if it can apply a large enough force, but it may require a large workspace area for the rotation. We are interested in multi-robot strategies that can overcome such limitations.

We now present a robust, implemented reorientation protocol which relies on the ability of our robots to execute *push-tracking* motions. The protocol is online, does not rely on *a priori* geometric models, is SPMD, asynchronous, and requires no communication between the agents.

For this protocol (called ASYNC-ONLINE), two robots suffice. Assuming that the robots begin in contact with the object, the following algorithm is executed by each robot asynchronously and in parallel to achieve the desired reorientation.

[4] This strategy can be implemented by a force that forms an acute angle on the contacting edge, outside the friction cone. This is similar to hybrid control [RC81] which would be used for a dexterous hand [Rus92].



(a) *(b)*

Figure 6: (a): Reorientation by one robot executing pushing-tracking. (b) A system of two robots reorienting a couch. The robot motions are shown in an object-centered frame. Each robot executes a pushing-tracking motion. Robots recognize when they reach the end of a face by breaking contact, and execute a spinning motion to turn around and reacquire contact.

Each robot:

1. (push-track) until contact break or impediment
2. if contact break then (spin)
3. if impediment then (graze)

The intuition is that the robots try to maintain the push-tracking state. When the end of the face is reached, the *spinning* motion is employed to reacquire contact (see Figure 6b). When an impediment is encountered, the robot executes a guarded move near-parallel to, but towards the face of the object, effectively *grazing* the object. Graze terminates when the robot recontacts the object or when it detects that it has traveled past the end of the face. Hence, graze terminates in (I) reacquiring the object at a contact with a longer moment arm, or (II) missing altogether. (I) is detected with a guarded move. (II) is detected using a sonar-based wall/corner detection algorithm of [JR93]. When (II) occurs, the robot executes (spin). The composition of these three circuits constitutes the circuit for a reorienting robot. ASYNCH-ONLINE consists of two reorienting robot circuits.

We have executed this protocol on our robots to reorient couches, file cabinets, and large boxes, sometimes through more than three complete revolutions (> 1080 degrees) .

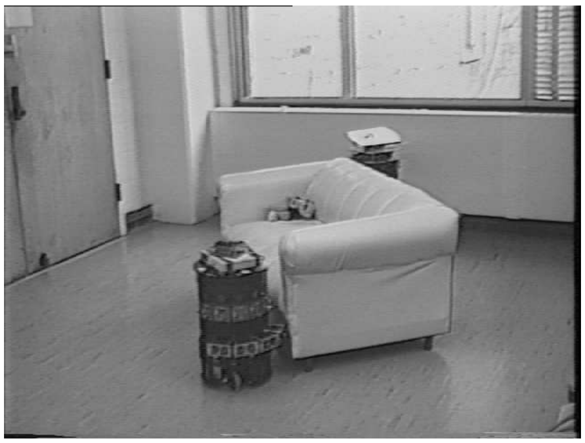


Figure 7: Two mobile robots cooperating to reorient a couch: a snapshot taken from a couch reorientation experiment.

This algorithm depends on the robust implementation of `(push-track)`. This is realized as the sequential composition of `(align)` and `(prim-push)` (defined in Section 5.1). `(align)` is invoked first to orient the robot to the face and to choose the correct pushing direction. Then, keeping that heading fixed, `(prim-push)` is invoked.

Figure 8 describes in detail the `(align)` circuit. We have run hundreds of experiments in which the robots reliably execute `(align)` asynchronously and in parallel. We believe that reliability and ease of reuse of our virtual orientation sensor come from its performance guarantees. In [JR93] we prove that `(align)` is robust and reliable in that for the geometry of our robots, the sensed angle is always accurate to within three degrees. Our experiments provide empirical evidence that these guarantees transfer when `(align)` is immersed in different fashions.

Similar analyses can be carried for the other circuits used by the reorientation protocol (*e.g.*, `(spin)` and `(graze)`.)

5.2.1 Supermodularity of Reorientation

Unlike `(prim-push)`, `(push-track)` is often supermodular. The reason is that when relocated, `(push-track)` adapts to the local surface and chooses a pushing direction (by using `(align)`) that imparts the correct torque (see Figure 9). The relocation of the `(push-track)` circuit from configuration A to B eventually results in the same net torque on the object, as B will slide to B$'$. This assumes that the robot slides. If the robot does not slide, then `(push-track)`

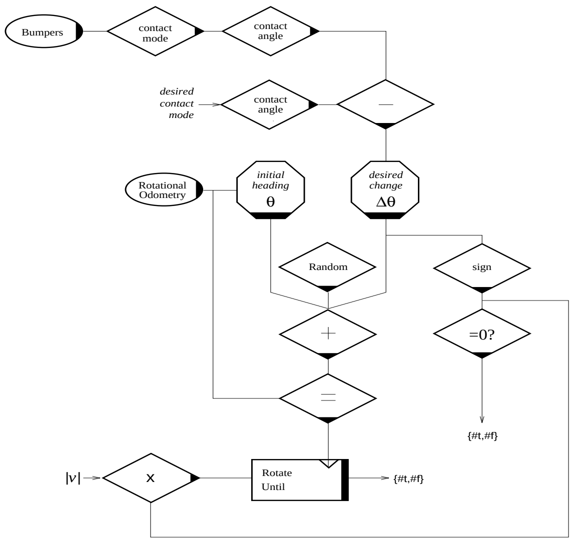


Figure 8: Circuit for `(align)`. The robot has a ring of contact bumpers and the goal is to position the robot so that the two front bumpers (for example) are compressed. The robot reads its bumper configuration and either terminates, rotates clockwise, rotates counterclockwise, or randomly repositions itself on the face of the object depending on the bumper values. If the two front bumpers are compressed it terminates. If a subset of the right bumpers are compressed it rotates counterclockwise. If a subset of the left bumpers are compressed it rotates clockwise. If the robot fails to align after several rotates, it reacquires contact by randomization.

may not apply the correct torque. This only happens when the friction between the robot and the object is such that no sliding is possible on the object surface. `(push-track)` can be extended to be completely supermodular always in the following manner. An independent sensor can be added to detect the situation where the robot applies a force that is within the friction cone at the point of contact. `(graze)`, which moves the robot without contact, can then be invoked to place the robot to B$'$, for a better lever arm.

Let us consider now the circuit ASYNC-ONLINE. ASYNC-ONLINE consists of `(push-track)`, `(spin)`, and `(graze)`, replicated on two spatially separated robots. ASYNC-ONLINE is more supermodular than `(push-track)`: it is guaranteed to reorient the object no matter what the starting configuration of the two robots is [RDJ95]. ASYNC-ONLINE is thus more supermodular than `(push-track)` since there are no constraints on where this circuit can be relocated, so long as it is on the body of the object.

We can quantify the supermodularity of a circuit by

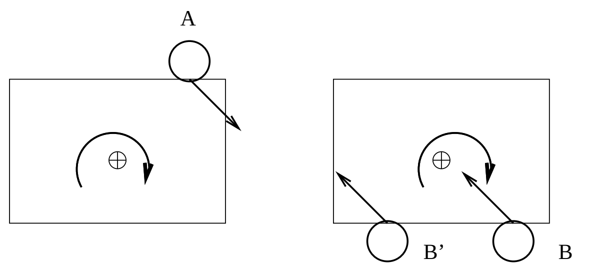


Figure 9: The supermodularity of the `(push-track)` circuit. In the left image, the robot A executes `(push-track)` and imparts the torque shown in the figure. The right image shows `(push-track)` relocated to B. Eventually, B will slide to B′ resulting in the correct torque.

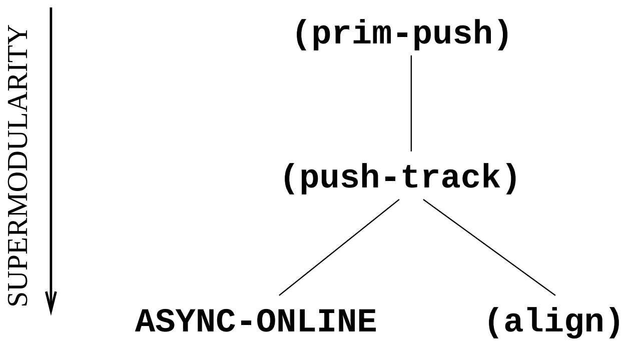


Figure 10: A partial order with respect to supermodularity. The arrow indicates "more supermodularity". `(prim-push)` is not supermodular. `(push-track)` is often supermodular, and `(align)` and Async-Online are always supermodular.

the fraction of configuration space where the circuit can be relocated and still function correctly. In our forthcoming work we will characterize this property algebraically [DRJ96]. Here we develop the following partial order on supermodular circuits. We have given examples of circuits that are not supermodular at all (such as `(prim-push)`), circuits that are often supermodular (such as `(push-track)`), and circuits that are always supermodular (such as `(align)` and Async-Online). The partial order on these circuits with respect to supermodularity is shown in Figure 10.

5.3 MPMD Manipulation

Section 5.2 presents an implemented and tested SPMD manipulation protocol. We now present an MPMD protocol called the *Pusher/Steerer system*. A detailed description and analysis of this system is given in [BJ95, Bro95].

The system consists of two robots. Each of the two robots executing this protocol either takes on

the role of the **Pusher**, in which

1. Torque-controlled translations push the object in front of the robot,
2. the robot follows the object by continually turning to align its front bumpers with the rear face of the object (the rotational and translational motions here are decoupled and occur in parallel), and
3. the robot does *not* know the path that the object is supposed to follow,

or the role of the **Steerer**, in which

4. The robot knows a path that it is supposed to follow,
5. the robot is translationally compliant (it controls the heading of its wheels, but does not control their rate of rotation), and
6. the robot moves forward as a result of being pushed by the object (which is itself being pushed by the Pusher).

Figure 11 shows two robots moving a rectangular object through a circular arc.

Now, it is sometimes possible to navigate an object in a straight line or along a circular arc using a single robot. If the object's center of friction (COF) is known and fixed, and the robot has little control error, then the robot can plan a control strategy to manipulate the object along a desired trajectory with acceptably small error. However, if the location of the COF is not known precisely, or changes with time (generally the case), then the robot cannot simply plan and execute a reliable trajectory, but must continually sense the object's relative orientation and position and compensate for drift. On the other hand, the forces exerted by the robots in the two-robot Pusher/Steerer system can constrain the object reliably in the presence of greater uncertainties, *e.g.*, in the location of the COF, and under coarser control and sensing.

In his monograph on information invariants [Don95], Donald claims that the spatial distribution of resources has a critical effect on the capabilities of a system. The Pusher/Steerer system validates that claim. Consider a *single-robot* manipulation algorithm such as, [LM94]. As implemented on the Cornell mobile robots, the execution system consists of the following resources, each of which can be represented as a circuit:

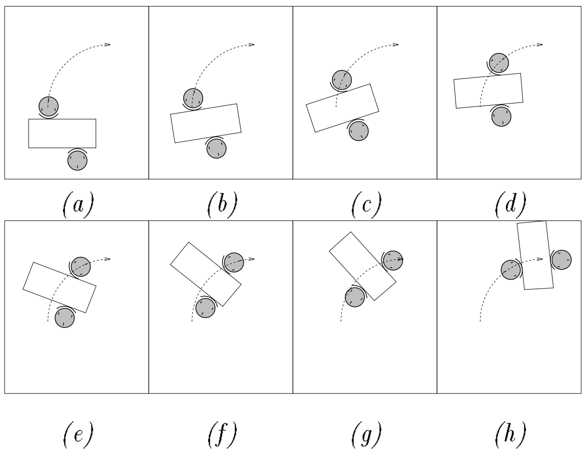


Figure 11: This series of figures depict a box being guided through a 90 degree arc by a steering robot (in front, following the arc), and a pushing robot. The box begins with its front and rear faces approximately perpendicular to the path. In *(b)* and *(c)*, the box rotates in the wrong direction, due to poor initial placement of the Pusher relative to the Steerer. By *(d)*, the Pusher, with no model of the box or the path and with no communication, has compensated for the poor initial configuration. By *(h)*, the box has traversed the arc and rotated until its front and rear faces are approximately perpendicular to the path.

- a *pushing* primitive, `(prim-push)` (see Section 5.1);
- `(align)` (see Section 5.2);
- a *steering* primitive, `(steer)`; and
- *a priori* path information.

5.3.1 Supermodularity in the Pusher/Steerer System

The Pusher's (entire) control system is obtained by the *parallel* composition of the circuits `(align)` and `(prim-push)`. The robot skill represented by the circuit `(push-track)` used in Section 5.2 consists of the *sequential* composition of `(align)` and `(prim-push)`. Thus the Pusher is equivalent to a single pushing robot (`(SPR)` in Section 5.1). This constitutes another example of supermodularity: here, the (vertex) immersion for the `(align)` circuit is the same as in the reorientation circuit, but the graph permutation is different.

The Steerer's (entire) control system consists of the `(steer)` circuit and path information.

If all the circuits listed above are co-located, that is, implemented on a single robot, and if the `(align)` circuit and the `(prim-push)` circuit are composed in parallel, then the resulting supercircuit is sufficient to implement a single pushing robot that can, for example, push a couch along a path. When the circuits are distributed across two robots—broken up as described above—then neither robot alone can manipulate a couch, but both robots together can manipulate large objects dexterously along a path. In addition, the resulting distributed system reaps benefits in terms of greater controllability and reduced sensing and knowledge requirements.

5.3.2 Role Trading

In the Pusher/Steerer system, the manipulated object sits between the robots. One feature of this system is that there is *no direct communication* between the two robots; they interact with *indirect* communication through the mechanics of the robots-and-object system. The configuration is conceptually similar to a rear-wheel-drive automobile that has been sliced into three sections: the rear wheels push the passenger compartment forward in a direction determined by the front wheels. The challenge to this configuration is that the pieces *are* separate—the robots have to be programmed to allow flexible trajectory following, while keeping the object between them. The advantage is that the robots can *trade roles*—the Pusher can become the Steerer and vice-versa. This increases the flexibility of the protocol by allowing such maneuvers as the "back-and-fill" that automobile drivers use for turning cars around on narrow roads. It also permits the robots-and-object system to reverse direction without the robots having to move to different faces on the object.

Role trading adds flexibility to the system but doubles the resource requirements. Each robot has to be equipped with all the circuits necessary for the Pusher role and for the Steerer Role. A system so equipped it is supermodular, so long as the Pusher and the Steerer have the same direction of action.

We have performed over 100 manipulation experiments using the Pusher/Steerer protocol running on several pairs of Cornell mobile robots. In these experiments, boxes and similar objects of varying size, mass, mass distribution, and material properties were manipulated along complicated paths up to 50 feet in length. On the basis of these experiments, along with others which are described in [Bro95], we have observed the system to be quite robust in practice. In each case, the code that each robot executed remained the same—there was no recoding or "tuning" for different objects or paths. Additional experiments using on-line navigation methods (human guidance in one case and visual landmark recognition in another) have demonstrated the flexibility of the system.

Despite conventional wisdom regarding the complexities of programming a multi-robot system, a key feature of the Pusher/Steerer system is its ease of use—the actual robot code is simple and elegant, and yet there remains great flexibility in methods of path specification.

6 Infrastructure

The programs that implement our distributed manipulation strategies were easy to generate and are robust. In this section we discuss our development infrastructure.

We begin with two architectural decisions that we first reported in our paper "Program mobile robots in SCHEME" [RD92]. For details see [RD92]. These decisions concern:

1. Why extend a general-purpose programming language instead of using or inventing a special-purpose robot language such as ALPHA [Gat91], or the behavior language [Bro90] ?

2. Of the general-purpose languages, why Scheme?

We regard robots as computers that can exert forces. In particular, we are interested only in the case these forces are external to the robot system—this excludes devices like clocks from being considered robots. More specifically, we focus on devices where these external forces are programmable: this excludes devices like VCRs, which are electronically, but not mechanically, programmable. Most specifically, we are interested in using these programmable external forces for manipulation tasks—that is, to rearrange the environment external to the robot (*e.g.*, to move furniture, perform assemblies, sort parts, *etc.*). Naturally, such robots typically use internal state to encode assumptions, models, and expectations about the external physical world. Equally often, sensors are employed not only to build such internal state but also to compensate for errors in actuation and *a priori* models.

As such, the internal state and control strategies can range from very simple to arbitrarily complex. The language must support the following operations:

1. access to effectors;

2. access to sensors;

3. ability to satisfy real-time constraints;

4. synchronization and communication between processes.

Since robot control algorithms can be quite general we chose a general programming language in which to express them and provide architectural support for the points above as follows:

(1) and (2): library calls

(3) and (4): light-weight processes (threads) implemented using continuations. Locks and condition variables provide synchronization.

Automatic memory management with garbage collection of a fixed physical address space (no virtual memory).

The last point yields a fixed upper bound for garbage collection delays, permitting rigorous reasoning about real-time programs.

This section describes how well our design decisions worked, and their implications for the infrastructure of manipulation protocols. We will describe domain constraints that must be satisfied to makes the use of SCHEME practical. The programming infrastructure has resulted in a particular set of manipulation idioms which permeate our code. We will describe how, using higher-order functions, we can implement termination predicates, control parameter caches, action cues, and event-based exception handling.

6.1 Mobot Scheme: The Software Logic Analyzer

In Sections 5.1–5.3 above, we described several distributed systems in which a team of mobots cooperates in manipulation tasks. The programs that implement the strategies for *Pushing*, *Reorientation*, and *Pusher/Steerer* are written in MOBOT SCHEME, a customized Scheme48 designed by Jonathan Rees for the Cornell Mobile Robots [RD92]. MOBOT SCHEME code for Protocol 2Mθ (Section 5.1) is shown in Figure 12. The most interesting feature from a development standpoint is the distributed nature of the programming environment. The robots act as Scheme servers; our workstations run a Scheme48 byte-code compiler and maintain serial connections to the robots by wire or by radio modem. The serial connection is used only during debugging—the robots are otherwise completely autonomous, possessing complete Scheme48 virtual machines on board.

A typical robot experiment session consists of interacting with the robot by typing on the workstation. The user works in an interactive SCHEME environment, usually running inside GNUEMACS. The

```
Both Robots Execute:

(define (1-M-theta)
  (let ((initial-angle (measure-theta)))
    (let repeat ()
      (if (positive?
            (push 'until
              (lambda (theta)
                (if (not (approx=? theta
                                   initial-angle))
                    (sign (- theta initial-angle))
                    #f))))
          (translate-left)
          (translate-right))
      (repeat))))
```

Figure 12: Protocol 2Mθ: code for two mobile robots with relative orientation sensing capability to execute the straight-line pushing task. This code is an implementation of the circuit shown in Figure 2

user has access to both the workstation facilities (*e.g.*, graphics, sound, filesystems), and the mobile robot. Interaction with the robot begins by invoking the mobot **read-eval-print** loop. Scheme forms are typed interactively at the workstation, which compiles them and sends them to the robot for evaluation. The reply is interpreted by the workstation and displayed on the screen. Therefore, the **read** and **print** steps occur on the workstation, while the **eval** is performed by the robot. While a program is being executed, the serial communications link between the robot and workstation is not required. It can be disconnected at anytime, and the robot will continue to run independent of the workstation. Of course, debugging information cannot be displayed during this time, and the robot's program cannot be altered. But both of these activities may resume any time the communications cable is reconnected. The design of the programming environment also permits remote procedure calls to take place between the robot and workstation, in either direction. The end result is that the full debugging facilities of the workstation are available for development, and yet the robot remains autonomous, able to execute programs untethered.

Our choice of MOBOT SCHEME for development reflects a minimalist approach to multi-robot programming. As a programming language, SCHEME is considered minimal. SCHEME comprises of a small set of functions that enable the programmer to build layers of abstraction by evaluating lambda expressions and remove layers of abstraction by applying a procedure to its arguments. The entire language definition, including its history, code examples, macro documentation, formal syntax and semantics, and extensive bibliography fits into 55 pages [CR92]. Scheme is a modern high-level functional language with automatic memory management, first-class functions, block structure, static scoping, closures, polymorphism, and dynamic typing.

Early manipulation programs for commercial anthropomorphic robot arms were written in a Basic-like language which lacked virtually every desirable feature of modern programming languages.[5] More recently, manipulators and mobile robots have been programmed in the C language.[6] The application of languages like Scheme, Lisp, or ML to robot control may have been primarily hindered by one particular feature: automatic memory management. Systems with this feature require periodic garbage collection, which often stalls or at least slows other processing. Since robot programming was assumed to require real-time response, garbage collecting systems were avoided. The reasons were: *(1) Robots need programs which meet real-time constraints,* and *(2) A language that garbage collects can introduce arbitrarily long delays.*

For many robots and many tasks, these reasons are not valid. First, many robots execute tasks which do not require real-time responses to sensory input. For example, most mobile robots do not need to dodge traffic on busy streets or play tennis. In particular, much manipulation work can be modeled as "quasi-static", meaning that inertial effects are negligible. When this is the case (largely because speeds are low and friction is significant), objects in the task stop moving as soon as the robot stops. And so the robot may "stop and think" for arbitrarily long periods without upsetting the task. Thus, pausing for garbage collection does not break the strategy, but instead merely slows it down.

Now, there are quasi-static tasks in which the robots cannot pause for arbitrary periods of time. In a dynamic environment, a robot may have to respond to conditions (*e.g.*, the presence of a human being, or the sudden appearance of an obstacle) in the world apart from the actual manipulation. (The interaction

[5]The most common example is the **VAL II** language which runs the Unimation Puma series of manipulators.

[6]The Zebra Zero manipulator arm is supplied with a C library of robot control functions, and many mobile robots are programmed in C due to the abundance of C compilers for small microprocessors. The Intel 80196 family and the Motorola 68000 family of processors, including embedded controller variations, are frequently used.

between the robot and the manipulated object may remain accurately described as quasi-static). We have found that garbage collection does not vitiate the robustness or even significantly degrade the performance of distributed manipulation protocols: a close examination of current mobile robots reveals that very few employ processors with virtual memory. In most cases, memory size is fixed, and is often small by workstation standards. As we pointed out in [RD92], with a constant amount of memory needing garbage collection, we can easily calculate an upper bound on how long this process could possibly take. Thus, we know in advance how long a pause may occur as the robot executes its strategy, and so we can predict just how effectively (in the worst case) the robot could react to changing conditions.[7]

A robot running a garbage collecting programming system can execute complex coordinated strategies in dynamic environments. Each cooperative strategy presented in this paper is an example. The programs that implement our strategies must of course access motors and sensors on our robots. In the next setion we describe this interface.

6.2 Higher Order Functions

The most critical features of our Mobot Scheme system are higher order functions, sensor-based control structures, and event-based exception handling. In this section we describe how using higher order functions we can implement termination predicates, control parameter caches, and action cues. Figure 12 is an example of how higher order functions are used in Mobot Scheme.

6.2.1 Termination Predicates

In our system, the robot programmer can write programs in which arbitrary termination predicates (functions) are passed as arguments to motion control routines. Intermediate layers of the motion control system build more complicated predicates out of these, and pass along the resulting functions to the next lower layer. As described in [RD92], we use higher order function to implement termination predicates.

[7]Moreover, robots like the Cornell Mobile Robots, which contain a loosely connected network of 9 or more (on average) processors, have a distinct advantage in that time-critical processes may be offloaded to a processor which does not garbage collect. Our robot's *impediment sensor*, for example, detects when a specified amount of current would have to be applied to the motors in order for the robot to continue moving. The processor which detects this situation and shuts down the motors executes a simple feedback loop at a high frequency, and does not garbage collect.

Termination predicates have been introduced in the pre-image motion planning framework of [LPMT84]. An extensive discussion of the relative power of these termination predicates in the context of motion planning may be found in [Erd84].

In the [LPMT84] framework, actions are pairs of the form $(v, \texttt{tp})$, where v is a velocity vector, and `tp` is a *termination predicate.* The motion continues in the direction v until the termination predicate returns `true`. The predicate, then, is a function of all of the information available to the robot through sensors and *a priori* models. In the idealized model, the predicate is evaluated continuously in the background.[8]

All of our motion control functions follow the same style for any effector (including, *e.g.*, a camera which may pan or tilt [Bro95]). Examples of Mobot Scheme code for various modes of translation (forward motion) are in Figure 13. We deviated from typical Scheme style by introducing keyword arguments which modify the effects of the motion. Hence, a set of defaults must be maintained, and there are `set-default!` and `get-default` functions for this purpose. Default values are used for any argument not given explicitly in the call to `translate`.

Our motion control functions represent sensor-based control loops. They initiate the robot's motion, which is usually controlled by another processor, and at the same time evaluate the appropriate termination predicate. The motion is halted when the termination predicate returns a value other than false. In order for access to the motors to remain completely general, many keywords exist, and may be used in arbitrary combinations. (See Figure 14.)

As indicated by the examples in the figure, keywords such as `'by` and `'vel` (for *velocity*) are followed by integer values. On the other hand, keywords such as `'until` are followed by functions. In the specific case of `'until`, a function of one argument is given, and the robot's motion stops when this function evaluates to something other than false. The argument to the predicate is supplied by the system when the predicate function is called and represents the current knowledge of the status of the motion. The motion controller (a separate processor) reports this status information to Mobot Scheme. In one example from Figure 13, we check to see whether the robot's bumpers have contacted an obstacle; in another, we check to see if sonar unit 2 reports a value that ex-

[8]On a real robot, the frequency at which the termination predicate is evaluated will of course critically affect its accuracy. However, the pre-image planning framework anticipated the general problem of control error from the start. Control error is already part of the input to the problem.

Example	Distance	Velocity	Termination Predicate
`(translate)`	*indet.*	*default*	`(impediment?)`
`(translate 'by 750)`	$750mm$	default	`(or (impediment?) (lambda (status) (stopped? status)))`
`(translate 'vel 220)`	*indet.*	$220mm/s$	`(impediment?)`
`(translate 'by 2000 'vel 400)`	$2m$	$400mm/s$	`(or (impediment?) (lambda (status) (stopped? status)))`
`(translate 'until (lambda (status) (not (zero? (read-bumpers)))))`	*indet.*	*default*	`(or (impediment?) (lambda (status) (not (zero? (read-bumpers)))))`
`(translate 'until (lambda (status) (> *threshold* (read-sonar 2))) 'by 640 'vel 450)`	$640mm$	$450mm/s$	`(or (impediment?) (lambda (status) (or (> *threshold* (read-sonar 2)) (stopped? status))))`

Figure 13: MOBOT SCHEME motion control examples using the `translate` function. For simplicity, only three keyword arguments are shown. Any or all of the valid arguments may be combined, however. When no distance argument is given (using the `'by` option), the distance the robot will travel is indeterminate. When a distance is specified, the termination predicate consists of a check for the `status` of the motor controller, which is triggered by checking odometry. Note that the system always checks for the `impediment` condition. The action taken when `(impediment?)` returns true is to invoke the impediment handler (see Section 6.3).

ceeds `*threshold*`.

Keyword	Description	Default value
`'by`	distance to travel (mm)	*none*
`'vel`	velocity (mm/s)	300
`'accel`	acceleration (mm/s^2)	225
`'dir`	direction to travel (forward is positive)	1
`'torque`	motor torque limit (in obscure units)	180
`'until`	termination predicate	`(impediment?)`
`'on-exit`	termination action	`(stop)` *halt the robot*

Figure 14: Motion control system keywords. The keywords may be used in arbitrary combinations, and each has a reasonable (and changeable) default value.

The default termination predicate, `(impediment?)`, reports whether the robot's motors are stalled[9] due to contact with an immovable obstacle. The action taken when an impediment is detected is specified by the impediment handler, because the situation is treated as an exception (see Section 6.3.) The system always checks for the impediment condition. Since the user may supply several other terminating conditions (such as a specified distance traveled, or some other arbitrary predicate), the system constructs, at run time, the proper predicate to use.

6.2.2 Write-through Cache

We may use higher-order functions to modify the low-level behavior of individual processors in the robot while still maintaining the single Scheme processor abstraction at the user level. The mechanism we use is a *write-through cache*, which can be implemented to be transparent to higher levels of code. Its sole utility is to boost performance, and works as follows.

Each robot contains, on average, more than nine individual processors, each of which is directly connected to a small number of sensors or motors. This heterogeneous collection of computers are programmed to make each sensor or motor "smart". As much processing as possible is done locally, with every

[9]The level of motor current that must be exceeded to meet the stalled condition may be set by software; an additional keyword argument, `'torque`, exists for this purpose.

sensori-motor module providing a high-level interface to the rest of the robot. Consequently, the robot's critical system parameters are distributed across many processors. When a Scheme program accesses a sensor or motor, at the lowest level a form of remote procedure call (RPC) is executed to the processor which controls the hardware of interest. The interprocessor communication and RPC are of course hidden to higher levels of Scheme programs, such as the the user programming level. Each time an RPC occurs, many parameters may be needed, which increases the amount of communication necessary and slows down the system. As a result, most processors store their own state variables, and the remote (calling) machine needs only pass along the changes in state.

We now present a brief case-study of using Mobot Scheme to tune a control library, thereby improving real-time performance. In our initial library, any Scheme function invoking a wheelbase motion would first set each parameter in turn and then finally start the motion. This was inefficient. We observed that almost always the parameters remain the same through many successive motions. Therefore, we decided to cache them on the Scheme computer using a standard write-through cache. We implemented a simple macro, `cache!` (see Figure 15), which accepts as its argument the function used to set a parameter. It modifies this function to use the cache, and also constructs a function to read the current value of the parameter from the cache. Its use is illustrated in Figure 16. What is interesting is that even this kind of low-level optimization may be done in Mobot Scheme, using higher-order procedures.

The `cache!` macro takes two arguments, the function which sets the parameter to be cached and the table to use. Two things happen: (a) the set function is modified to set the current value in the cache table, and then the original set function is called, and (b) a get-value function (thunk) is returned. Each entry in the cache table has a key and an entry. The key is the new set function, and the entry is always a pair `((value) . original-set-fcn)`. The value is kept in a one-element list so that it will be null if no value has been set. The original set function is kept around so that (i) we can get it back if we need it, and (ii) we can call it to write the cache out (to force a write).

Because we used a modern general purpose programming language with higher order functions, implementing `cache!` was easy. The cache is transparent to the low-level layers of the robot control system, which performs the necessary communications

```
(define *wb-state-table* (make-table))
(define (write-wb-state)
  (write-cache *wb-state-table*))
(define translate-speed
  (cache! set-translate-speed!
          *wb-state-table*))
(define translate-accel
  (cache! set-translate-accel!
          *wb-state-table*))
(define translate-torque
  (cache! set-translate-torque!
          *Wb-State-Table*))
```

Figure 16: Examples of using a simple write-through cache, implemented using a macro and higher order functions.

between processors, and also to the higher levels of the system, such as the user programming environment. The cache may be trivially removed in the event that we decide not to trust the remote processor to retain its state between successive motions (*e.g.*, due to a suspected faulty piece of hardware).[10] On the other hand, the macro is general enough to be used in other caching applications. Our `cache!` illustrates that standard software design techniques are also applicable to robot programming. We are far from the days of programming a single 8-bit processor in assembly language in order to make our robots move.

6.2.3 Action Cues

Our final example of higher order functions lies in the interface to the Cornell Mobile Robots' set of pushbuttons which, along with multi-colored LEDs, reside on top of the robot for interaction with users. Especially when operating untethered, the information displayed by the lights and entered via the buttons provides useful communication between the robot and the researcher. At the user programming level, the `set-button-action!` procedure takes two arguments: the first identifies a particular button on the robot, and the second is a function of no arguments (technically a *thunk*). When the appropriate button is pressed, the associated function is executed in its own thread, concurrently with whatever other threads (processes) are running on the robot. The mechanism

[10] On the Cornell Mobile Robots, the Scheme processor can detect when the wheelbase processor resets, and a complete cache write automatically occurs after those events, along with a warning message displayed on the workstation.

```
(define-syntax cache!
 (lambda (exp rename compare)
    (let ((set-fcn (cadr exp))
          (state-table (caddr exp)))
      `(let ((original-set-fcn ,set-fcn))
         (set! ,set-fcn (lambda (x)
                          (let ((value (table-ref ,state-table ,set-fcn)))
                            (if (or (null? (car value)) (not (equal? (car value) (list x))))
                                (original-set-fcn x))
                            (table-set! ,state-table ,set-fcn (cons (list x) (cdr value))))))
         (if (not (table-set! ,state-table ,set-fcn (cons '() original-set-fcn)))
             (begin
               (set! ,set-fcn original-set-fcn)
               (error "Cache error: table full?" ',set-fcn ',state-table))
             (lambda ()
               (let ((value (table-ref ,state-table ,set-fcn)))
                 (if (null? (car value))
                     (error "Cache error: the referenced value has not been set." ',set-fcn ',state-table)
                     (car (car value))))))))))
(define (write-cache cache-table)
 (table-walk
   (lambda (set-fcn value)
     (if (not (null? (car value)))
         (apply (cdr value) (car value))))
   cache-table))
```

Figure 15: A simple write-through cache implemented using a Scheme macro.

allows complete freedom in deciding what the buttons should do. The flexibility is provided by the passing of an arbitrary Scheme function directly into the next lower layer of the sensor control system. In fact, many of our sensors are programmed in this manner, including the pushbutton bumpers that detect collisions with objects and measure the relative orientation of object faces with respect to the robot's heading. In writing a program which polls the bumpers, we first write a function which, when a bumper button is pressed, simply updates a global dynamic variable. The polling process then reads this variable to determine the current state of the bumper sensor.

6.3 Collisions as Exceptions

We say a *contact change* occurs at time t when the robot is in contact with a different number of surfaces before t than after t. We say the contact change is *positive* when the number of contacting surfaces increases. Hence, a contact change is an *external event* to the robot (in other words, ground truth)—observable to an outsider but potentially undetectable by the robot.

An impediment, on the other hand (see Section 6.2), is an *internal event* signaled by and for the robot itself. We define a *collision* as a special kind of event that convolves the (internal) impediment and the (external) contact change. More precisely, a *collision* is the special case of an impediment caused by a positive contact change.

Unexpected collisions should be treated as exceptions. The idea is straightforward—we can either fill our programs with conditionals which check for unexpected situations, or we can invoke the exception handler and make our programming task that much easier. Also, we must be able to easily modify this behavior for those circumstances under which the programmer fully expects that a given situation might occur, and wishes to trap it so that the program can continue to execute, taking appropriate action.

A global variable called `impediment-handler` is defined as a function which is called when the motion control system detects the impediment condition. The default value for the handler is a function which first stops the robot's motors and then calls the system exception handler, presenting all relevant information to the user, and leaving them in the symbolic debugger (see Figure 17 Example 2). From within the debugger, the user may continue (*e.g.*, after removing the offending obstacle from the robot's path), abort, or perform almost any other action, such as recompiling a piece of code, or using the inspector to examine the stack.

So, the default reaction to a collision is for the robot to stop and the user to be presented with status information displayed through the debugger. In other words, the situation is treated as any other run-time error would be, such as division by zero or a type mismatch. It remains to be explained how the user may write a program which traps expected collision errors.

```
;;
;; Example 1.
;;
;; Move until the bumpers detect a collision
Lily> (translate 'until
        (lambda (s)
          (not (zero? (read-bumpers)))))
#t
;;
;; Example 2.
;;
;; Move forever monitoring the motor current
;; against a current threshold.
;; Here we grab the robot and force it to stop,
;; which invokes the debugger that allows us
;; us to examine variables, redefine functions,
;; etc. :a aborts out of the debugger.
Lily> (translate)
Error: Impediment
       translate
       100
(Debugging job number 25)
Lily-> :a
;;
;; Example 3.
;;
;; Expected collisions can be trapped. We use
;; a tolerate-impediments wrapper around the
;; function that might generate a collision.
;; The impediment handling system will stop
;; the robot and the motion function return
;; 'impediment.
Lily> (tolerate-impediments (translate))
'impediment
```

Figure 17: Three examples of impediments handling. It is straight-forward to substitute an arbitrary function for the impediment handler.

The mechanism we use replaces the impediment handler with a different function while a collision is a predicted possibility and should be trapped. In our experience, the most common alternative to the function which calls the exception handler is a function which simply returns the symbol `'impediment`. The user program needs only check for this symbol to be returned by a motion control routine in order to branch on whether a collision occurred or some other situation terminated the motion. Because we use this mechanism frequently, we have defined a macro, called `tolerate-impediments` (see Figure 17 Example 3), which replaces the default impediment handler with a function that returns `'impediment` while the body of the macro is being executed. Consequently, one may modify any motion control function call by wrapping `tolerate-impediments` around it, as shown in the example.

The flexibility of MOBOT SCHEME permits the use of this mechanism, which can be seen in virtually every manipulation program our robots run. All we have done was to adapt a traditional technique to our application of programming robots. Our programs are short and simple as a result of this technique.

7 Architecture

When the minimalist philosophy is applied to robot manipulation algorithms it yields protocols with surprisingly low resource consumption—that is, the resulting protocols often dispense with a resource (such as sensing, communication, or geometric models) that a naive analysis would have predicted was essential [DJR93, DJR94a, RDJ95, DJR94b]. When applied to the programming environment and architectural support for robot manipulation, minimalism results in a development and execution system that is spare and lean—in other words, something more like Scheme and less like C++ (Section 6). In this section we discuss the use of our Scheme-based minimalist infrastructure in supermodular architectures.

The greatest challenge for supermodular architectures is to support code distribution across multiple spatially separated robots. It is difficult to ensure code portability across robot architectures, and the result of the execution in distributed systems of multiple robots is even harder to predict. Even for single robot systems that are mechanically similar (such as our robots TOMMY and LILY), it is not trivial to share code, as this often entails adjusting constants that involve mass, gear ratios, *etc.* Furthermore, for manipulation tasks, the output of a distributed system of robots often depends on the number of robots and their relative positions.

Consider, for example, a system in which a robot pushes an object to effect a desired translation and the circuit implementation of this system we describe in Section 5.1. This system is modular but not supermodular. The reason is that when distributed on multiple robots, the output of the system depends intimately on the location of the robots. To see this, imagine a two robot system consisting of the original robot and a second robot that acts on a face of the object opposite to the first robot. The presence of the second robot interferes with the action of the first robot (*e.g.*, it may cause the box to rotate). This is not a supermodular system; in other words `(prim-push)` (Section 5.1) is modular but not supermodular.

A contrasting example is the reorientation task. Here, one robot alone can cause reorientations by executing `(push-track)` (Section 5.2), but this may require a large workspace area. When the same code is distributed over multiple robots, the output of the system is the same (*e.g.*, the object rotates), but perhaps at a faster rate. The resulting circuit, Async-Online is completely supermodular. In our hierarchy of supermodular examples, Async-Online is more supermodular than `(push-track)`, which is more supermodular than `(prim-push)`.

This is a supermodular system; in other words `(push-track)` is modular and supermodular.

Supermodularity has made it easy for us to share and distribute code across similar—but not identical—robots that operate in parallel. This is because the (abstract) principles of supermodularity translate into concrete benefits in terms of code recycling and predictability in distributed robot systems. By definition, supermodular circuits can be relocated to different physical locations and still function correctly. In addition, we believe that the performance guarantees of supermodular circuits transfer when the circuits are distributed.

The pushing vs. reorienting example described above shows that supermodularity and distribution impact architecture. We now discuss this connection with emphasis on the following questions:

1. *What architectural support is required to distribute the circuits over several agents?*

   We cannot begin to answer this question until we understand what it means to distribute a circuit over several agents. The installation and calibration of resources must be specified, and communications pathways across spatially separate locations must be available. (The precise definition of these operations is given in the information invariants theory [Don95].) But for a distributed circuit to achieve the same task, we must demonstrate that the embedded circuits effect the same strategy.

   For our particular domain of manipulation tasks, we must account for the mechanical interactions between the robots and their environment. On a robot such as Camel, whose bumper geometry is essentially a flat blade, the effect of pushing against an object may be different from that of, say, Tommy, whose bumper is in the shape of a semi-circle. For manipulation, then, we observe that compatible mechanical architectures are required when distributing a circuit across several agents. However, by transforming circuits into other equivalent circuits, we have "ported" our strategies without difficulty from Tommy to Camel[11] and vice-versa. The transformation operates at the circuit level, so in the final analysis we are still moving a (transformed) circuit from one robot to another.

   We have identified that the robot's mechanical architecture may limit supermodularity for manipulation tasks, because different strategies are required for different mechanical interfaces to the world. An important line of inquiry remains, which is to ask how the necessary architectural support for supermodularity varies with the task domain.

2. *What development environment best supports the authoring of supermodular circuits and protocols?*

   Our authoring environment is developed on top of an intrastructure (Section 6) that enforces minimal constraints on programming. The system we advocate is orthogonal both to the notion of modularity in cognitive AI and also to the horizontal decomposition of Subsumptionism. Specifically, our development environment is structured by the following categories: modular circuits, supermodular circuits, and strategies composed of supermodular circuits (or supercircuits).

   The first category comprises the modular circuits for the interaction between robots and objects and for navigation. Conceptually, these circuits can represent a wide range of robot skills of varying degree of abstraction and complexity. A circuit in this category is a single-agent circuit that is safe for relocation on other single-agent systems, but there are no "degrees of freedom": the

[11] Camel is a Cornell mobile robot that uses treads instead of wheels for locomotion.

circuit must be immersed identically. An example of a modular manipulation circuit in this category is `(prim-push)`.

The second category comprises the manipulation circuits that are often supermodular. In our hierarchy measured by the fraction of the configuration space that allows relocation, ASYNC-ONLINE and `(align)` are completely supermodular. `(push-track)` is often supermodular. A circuit in this category is safe for relocation even when the circuit is immersed in a different fashion. The "degrees of freedom" of this immersion are controlled by the kinds of permutations permitted in the information invariants theory (in the form of co-designation constraints.)

The third category comprises the circuits for manipulation strategies. A circuit in this category is a multiple agent circuit consisting of supermodular circuits. The resulting supercircuit may or may not be supermodular. Examples are Protocol 2Mθ, protocol ASYNC-ONLINE, and PUSHER-STEERER, which are supermodular.

3. *How can the architectural support and authoring environment be realized in such a way that the overall development system can also be regarded as minimalist?*

   The development environment we outlined above makes it easy to synthesize distributed manipulation strategies (*i.e.*, circuits for manipulation strategies) by combining and composing supermodular circuits. Since each supermodular circuit translates into lean code with performance guarantees, predictable parallel protocols can easily be generated by combing existing and trusted circuits. The resulting circuit strategy can be analyzed in the information invariant theory with respect to resource consumption. This analysis leads to equivalent minimalist circuits and trade-offs between different resources. The resulting circuit also translates into a statement about the minimal architectural support for the protocol: the circuit describes all the needed resources.

In summary, the architecture we advocate for authoring supermodular circuits arises from systematically distributing simple and parsimonious manipulation circuits on spatially separated robots. We argued that circuits authored in a supermodular architecture translate into code that is simple, reusable, portable, predictable, parallelizable, and "near-optimal" from the point of view of resource consumption [BBD+95]. We have experimental evidence for these properties—for example the reuse of `(align)` in the reorientation protocols and the Pusher/Steerer protocols. Similarly, we have performance guarantees. For example, for `(align)`, the performance guarantees transfer nicely once `(align)` in immersed in a different fashion—the "degrees of freedom" of the new immersion are controlled by the kinds of permutations permitted in the information invariants theory (in the form of co-designation constraints) and the niceness of the transfer is ensured by supermodularity.

Acknowledgements

This paper describes research done in the Robotics and Vision Laboratory at Cornell University. Russell Brown is one of the developers of the Pusher/Steerer system. We thank Jonathan Rees for developing the MOBOT SCHEME system. We thank Jean-Claude Latombe for his hospitality at the Stanford Robotics Laboratory.

Support for our robotics research was provided in part by the National Science Foundation under grants No. IRI-8802390, IRI-9000532, IRI-9201699, and by a Presidential Young Investigator award to Bruce Donald, and in part by the Air Force Office of Sponsored Research, the Mathematical Sciences Institute, Intel Corp., and AT&T Bell laboratories.

References

[ABN93] R.C. Arkin, T. Balch, and E. Nitz. Communication of behavioral state in multi-agent retrieval tasks. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 2, pages 588–594, Atlanta, Ga, 1993.

[BBD+95] K. Bohringer, R. Brown, B. Donald, J. Jennings, and D. Rus. Distributed robotic manipulation:experiments in minimalism. In *Proceedings of the International Symposium on Experimental Robotics*, Stanford, CA, July 1995.

[BJ95] R. Brown and J. Jennings. Manipulation by a pusher/steerer. In *Proceedings of Intelligent Robot Systems*, Pittsburgh, PA, August 1995.

[Bro85] R. C. Brost. Planning robot grasping motions in the presence of uncertainty. Carnegie-Mellon Robotics Institute technical report CMU-RI-TR-85-12, Computer Science Department and The Robotics Institute, July 1985.

[Bro90] R. Brooks. The behavior language user's guide. Technical report, Memo 1227, MIT AI Lab, 1990.

[Bro95] R. G. Brown. *Algorithms for Mobile Robot Localization and Building Flexible, Robust, Easy to Use Mobile Robots.* PhD thesis, Cornell University, Ithaca, NY, 1995.

[CFKM95] Y. Cao, A. Fukunaga, A. Kahng, and F. Meng. Cooperative mobile robots: Antecedents and directions. Technical report, UCLA Department of Computer Science, 1995.

[CR92] W. Clinger and J. Rees. Revised^4 report on the algorithmic language scheme. Technical report, Cornell University Department of Computer Science, 1992.

[DJR93] B. R. Donald, James S. Jennings, and D. Rus. Towards a theory ofinformation invariants for cooperating autonomous mobile robots. In *International Symposium on Robotics Research (ISRR)*, Hidden Valley, PA, 1993.

[DJR94a] B. Donald, J. Jennings, and D. Rus. Information invariants for distributed manipulation. *The First Workshop on the Algorithmic Foundations of Robotics, eds. K. Goldberg, D. Halperin, J.-C. Latombe, and R. Wilson*, pages 431–459, 1994.

[DJR94b] B. R. Donald, James S. Jennings, and D. Rus. Analyzing teams of cooperating mobile robots. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 1896–1903, San Diego, CA, 1994.

[DJR94c] B. R. Donald, James S. Jennings, and D. Rus. Information invariants for distributed manipulation. In *Proceedings of the 1994 Workshop on the Algorithmic Foundations of Robotics*, San Francisco, CA, 1994.

[Don95] B. Donald. Information invariants in robotics. *Artificial Intelligence*, 72:217–304, 1995.

[DRJ96] B. R. Donald, D. Rus, and J. Jennings. Quantifying supermodularity in dostributed manipulation circuits. In *forthcoming paper*, 1996.

[Erd84] M. Erdmann. On motion planning with uncertainty. Master's thesis, Massachusetts Institute of Technology, 1984.

[Gat91] Erann Gatt. Alpha:a language for programming reactive robotics control systems. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 1116–1121, 1991.

[Jen95] J. S. Jennings. *Distributed Manipulation with Mobile Robots.* PhD thesis, Cornell University, 1995.

[JR93] J. Jennings and D. Rus. Active model acquisition for near-sensorless manipulation with mobile robots. In *IASTED International Conference on Robotics and Manufacturing*, pages 179–184, Oxford, England, September 1993.

[LM94] Kevin M. Lynch and Matthew T. Mason. Stable pushing: Mechanics, controllability, and planning. In *Proceedings of the 1994 Workshop on the Algorithmic Foundations of Robotics*, San Francisco, CA, 1994.

[LPMT84] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, **3**(1), 1984.

[Mas95] M. Mason. Manipulator grasping and pushing operations. *International Journal of Robotics Research*, 5(3):53–71, 1995.

[Mat93] Maja J. Mataric. Kin recognition, similarity, and group behavior. In *Proc. of the Fifteenth Annual Cognitive Science Society Conference*, Boulder, Colorado, 1993.

[MS85] Matthew T. Mason and J. Kenneth Salisbury. *Robot Hands and the Mechanics of Manipulation.* MIT Press, London, England, 1985.

[Nor93] Fabrice R. Noreils. Toward a robot architecture integrating cooperation between mobile robots: Application to indoor environment. *International Journal of Robotics Research*, **12**:79–98, 1993.

[OY92] Yoshikuni Okawa and Ken Yokoyama. Control of a mobile robot for the push-a-box operation. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 761–766, Nice, France, 1992.

[Par94] Lynne E. Parker. *Heterogeneous Multi-Robot Cooperation.* PhD thesis, Massachusetts Institute of Technology, 1994.

[RC81] M. Raibert and J. Craig. Hybrid position/force control of manipulators. *Journal of Dynamic Systems, Measurement, and Control*, 102, 1981.

[RD92] J. A. Rees and B. R. Donald. Program mobile robots in scheme. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, 1992.

[RDJ95] D. Rus, B. Donald, and J. Jennings. Moving furniture with mobile robots. In *Proceedings of Intelligent Robot Systems*, Pittsburgh, PA, August 1995.

[Rus92] D. Rus. *Fine motion planning for dexterous manipulation.* PhD thesis, Cornell University, Ithaca, NY, August 1992.

[SB93] Daniel J. Stilwell and John S. Bay. Toward the development of a material transport system using swarms of antlike robots. In *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, pages 766–771, Atlanta, GA, 1993.