July 1990 Report No. STAN-CS-90-1323

# Protograms

by

**Eyal Mozes and Yoav Shoham**

**Department of Computer Science**

**Stanford University**

**Stanford, California 94305**

# Protograms

Eyal Mozes and Yoav Shoham

Department of Computer Science

Stanford University

June 8, 1990

Abstract

Motivated largely by tasks that require control of complex processes in a dynamic environment, we introduce a new computational construct called *a protogrum*. A protogram is a program specifying an abstract course of action, a course that allows for a range of specific actions, from which a choice is made through interaction with other protograms. We discuss the intuition behind the notion, and then explore some of the details involved in implementing it. Specifically, we (a) describe a general scheme of protogram interaction, (c) describe a protogram interpreter that has been implemented, dealing with some special cases, (c) describe three applications of the protogram interpreter, one in data processing and two in robotics (both currently only implemented as simulations), (d) describe some more general possible implementations of a protogram interpreter, and (e) discuss how protograms can be useful for the Gofer project [ 1]. We also briefly discuss the origins of protograms in psychology and linguistics, compare protograms to blackboard and subsumption architectures, and discuss directions for future research.

This work was supported by grants from NSF and AFOSR.

# 1 What are protograms?

The standard view in computer science of a program is as precise prescription of a course of action to be taken, a one-to-one input-output function. Indeed, any program is ultimately just that. Nevertheless, for various applications there are often more illuminating models of computation, models which facilitate programming, debugging and maintenance. Two facets of computational problems that, among others, have motivated new models of computation, are distributivity of processing and complexity of information. These are the motivations behind *blackboard systems* [2, 3], and behind some other models of computation. Broadly speaking, distributivity and complexity, and especially the complexity of processes involving continuously-varying parameters, are also the motivation for the new notion of *protograms*.

Protograms reflect the intuition that most activities are determined by several influences, which refine, reinforce, complement, and often contradict one another. This is true of our everyday behavior. We use a car to get to work because walking would take too long, and furthermore we use our spouse's car because ours is low on gas. We drive to the City fast, in order to make the concert, but not too fast, to avoid accidents and speeding tickets. When we learn how to ski we try to lean forward as instructed, but fear of falling tends to make us lean back; over time the fear subsides and becomes a very weak influence.

These multiple influences exist also in programming. We sort the data base of employees alphabetically, but place the manager, Zbigniew Zablowski, at the head of the list; we drive the robot towards the goal, but veer it away from obstacles. The standard view of programs requires the programmer to resolve these various influences and produce a single course of action. This is often feasible. In complex tasks, however, it can be very difficult, and that is where protograms are useful. They allow the programmer to specify only the individual influences — each represented by an abstraction of an action, allowing for a (continuous or discrete) range of concrete actions, and relative preferences within that range — as well as information about the relative importance of the influences. The protogram interpreter then uses that information to produce an unambiguous action description, which satisfies, to various degrees, the requirements of the individual protograms. The advantage to the programmer, besides exemption from the need to specify one global behavior, is modularity and incrementality: influences may be added and removed without having to write the program from scratch each time.

Consider in more detail the robot navigation task, and in particular a local

method such *as* the *artificial potential field method* [4].[1] In this procedure the robot is influenced by two imaginary forces — one attracting it to the goal, another repelling it from nearby obstacles. The movement of the robot is dictated by the sum of these two imaginary forces.

This procedure has been implemented widely in conventional languages, but it is possible to view it as an interaction between two protograms. The "goal" protogram recommends that the robot head directly towards the goal, and regards any other direction as less preferable to the extent that it deviates from heading towards the goal. The "obstacle" protogram recommends that robot head away from obstacles, and regards any direction as less preferable to the extent that it gets closer to an obstacle. The strength of the second recommendation is a decreasing function of the distance from the obstacle. The advantage of this view is that now we may add new influences. For example, an important deadline approaching may increase the strength of the "goal" protogram; or, if the robot is holding a mug filled with coffee, this may increase the strength of the "obstacle" protogram. We may also add completely new protograms (e.g. "stay at the center of the corridor"), and have their influence incorporated into the behaviour of the robot. Figure 1 illustrates the effect of different protogram combinations on such a robot.

The remainder of this article is organized as follows. In the next section we look more closely at protograms and the various ways they interact. In Section 4 we describe an experimental protogram interpreter that we have implemented. Then in Section 5 we describe two applications of this interpreter. In Section 8 we compare the protogram framework to other computational paradigms in AI, and discuss the connection to literature in psychology and linguistics. Finally, in section 9 we summarize the paper and describe our future work.

## 2 A closer look at protograms and interactions among them

Specification of a system of protograms consists of three parts:

[1] In robot navigation one distinguishes between local methods, which sense the immediate environment and act on this limited information, and global methods, which take into account all spatial information about the terrain. The former are typically fast but incomplete, the latter typically complete but slower. For thorough coverage of robot navigation see e.g. [5].

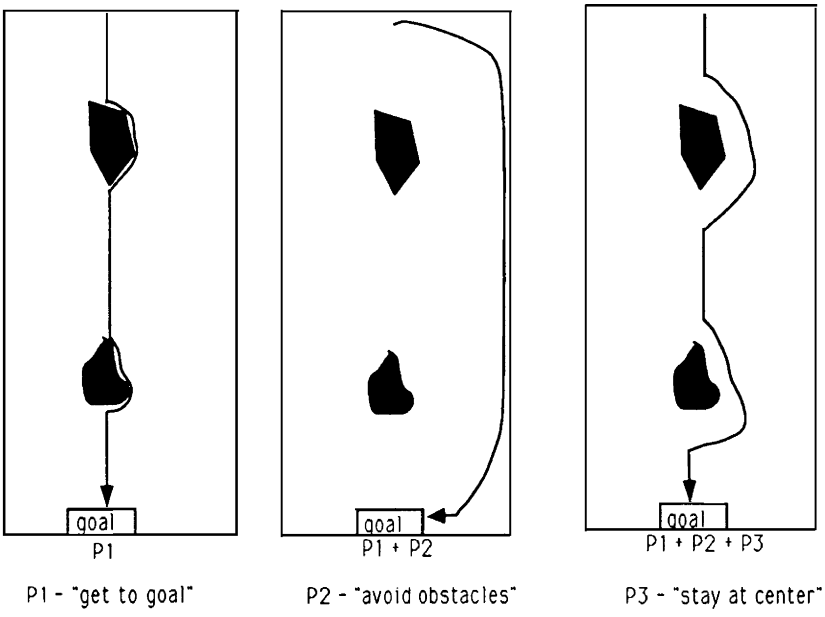


Figure 1: Protogram combinations in local robot navigation

1. For each protogram, a range of actions and preferences, as a function of the input. For example, for the "stay at the center of the corridor" (henceforth SCC) protogram, the range of actions is expressed by directions of movement, and a direction is more preferable the closer it brings you to the center from the current position (which is the input to the system); the ideal action for this protogram is moving directly towards the center.

2. For each protogram its relative "urgency" or "priority," again as a function of the input. For example, the urgency of the SCC protogram may be an increasing function of the distance to the center and, say, the fragility of the payload.

3. An arbitration procedure for trading off the deviations from the various ideals given the relative priorities.

The arbitration procedure is the most interesting component, the heart of the protogram approach. The complexity of the system is, in effect, pushed into this component, allowing other components to each handle an isolated part of the problem.

Let us look at some examples of protograms, demonstrating different possible types of interaction.

Consider a robot in an automobile factory, whose job it is to both assemble cars and haul in shipments of parts. The robot should constantly work on the assembly until a shipment arrives, at which point the robot should suspend the assembly task, haul in the shipment, and, when it is done, return to the assembly task. In effect, the robot is controlled by two protograms, in a fixed hierarchy of priorities, interacting by one simply overriding the other. Whenever the "haul in the shipment" protogram can act, it completely controls the robot's behaviour.

It is also possible to have a hierarchy in which a higher level does not completely override the lower one, but imposes constraints under which the lower level must optimize. For example, as we mention in section 1, you may want to sort a list of employees according to the two protograms "sort the list alphabetically" (henceforth SA) and "place the manager at head of list" (henceforth PMAH). PMAH is the higher-priority protogram. SA has a measure of deviation from the ideal, e.g. the number of pairs in reverse order, which it tries to minimize. The list is sorted alphabetically as far as possible, subject to the constraint of having the name of the manager at the head.

In both the above examples, the hierarchical order is fixed, i.e. the same protogram always overrides (or imposes constraints on) the other one in case of conflict. This need not be the case; it is also possible to have a system of protograms, interacting by overriding or by constraints, in which the hierarchical order can vary dynamically. An example of such a case is discussed in section 5.3.

As an example of a different type of interaction, consider again the robot navigation task, discussed in section 1. The different protograms (which, in this case, might be implemented as artificial potential field forces) operate in parallel, continuously determining the path of the robot; since they will generally disagree on the path, it must be determined by a compromise between them. Here the relative priorities of the protograms have to be expressed, not just as an ordering, but as a numerical ratio (which can be implemented by the relative strengths of the potential field forces), to determine their relative roles in the compromise. As mentioned above, the relative priorities may change dynamically (e.g. by a deadline approaching).

The above examples demonstrate three possible types of interaction between protograms, in increasing order of generality:

- Overriding: one protogram, at every moment, is used to determine the behavior of the system, ignoring the others.

- Hierarchical constraints: several protograms interact, and one has a higher priority. The other protograms must be satisfied as far as possible subject to the constraint imposed by the higher-priority one.

- Compromise among constraints: several protograms interact, and the behavior of the system is a compromise between their different instructions. This compromise can be based on either numerical or qualitative specification of the relative priorities.

# 3 Mathematical definition of protogram interaction

To integrate the above examples into a general framework, we propose a mathematical definition.

We assume that the action of the system is controlled by n parameters, $v_1$ to v,. Each $v_i$ is chosen from some set Vi, according to the instructions of the protograms.

The operation of a system defined by programs is expressed as a series of what we call "protogram-interaction cycles". At each such cycle, each protogram returns its instructions — its priority and an indication of its preferences about each $v_i$ — depending on the input. The arbiter then needs to find the $v_i$ values according to all instructions, and pass them as arguments to the procedure performing the action.

A "protogram-interaction cycle" is formally defined as a tuple $(V_1, V_2, \ldots, V_n, C)$, where $V_i$ are sets from which the $v_i$ are chosen, and C is a set of "protogram instructions,, , i.e. indications.

Each protogram instruction in C is a formally defined as a tuple $(P, S_1, S_2, \ldots, S_n)$, where P is a positive number (the protogram's priority) and each $S_i$ is a function $V_i \longrightarrow R$; i.e. $S_i$ states the degree to which each value of parameter i will satisfy the requirements of the protogram.

The result of the interaction is a tuple $(v_1, v_2, \ldots, v_n)$, s.t. for each i, $\sum_C (P * S_i(v_i))$ is maximized; i.e., the arbiter chooses a value for each parameter that maximizes the sum of satisfaction of all protograms, weighted by their priorities.

Note that each $V_i$ is an arbitrary set; usually, it will be a numerical value, but it may also be a tuple. This naturally expresses the fact that parameters of the action may be independent ( and should then be independently maximized), or may depend on each other. Each $v_i$ can be a cluster of interdependent numerical parameters — so the corresponding $S_i$ functions are

maximized for all of them together — and different $v_i$s are used for mutually independent parameters.

The three forms of interaction identified in section 2 generally fit into this scheme, though some assumptions are required for overriding and for hierarchical constraining:

- In the case of overriding, we need to assume that the actions recommended by the protograms are mutually exclusive, so that two protograms can never agree on the actions they want (e.g. our example of the robot in the automobile factory, in which one protogram always recommends working on the assembly task, and the other always recommends hauling in the shipment). Each protogram returns, for each parameter, an $S_i$ function that returns a constant high value (e.g. 100) on one of the $v_i$s, and a constant low value (e.g. 0) on all other $v_i$s. The highest-priority protogram will win, its desired action being performed.

- In the case of hierarchical constraining, we need to assume either that there are only two protograms, or that the priorities are fixed and the priority of the higher-priority protogram is higher than the sum of all the other protograms' priorities. The higher-priority protogram returns an $S_i$ function with a high value for some range of $v_i$s, and a low value for all other $v_i$s; no other protogram can use a lower value in its $S_i$ function. The value chosen will then be in the high-values range, and, within that range, the other protogram(s) will be optimised.

- In the case of compromise, the $v_i$ parameter has to be from some metric space (number, vector, etc.); and each protogram returns a "bell-shaped" $S_i$ function, with a high value on its desired value getting progressively lower as it gets further away. By maximizing $\sum_C( P * S_i(v_i))$, the arbiter reaches a value which is close to the desired value for each protogram depending on its priority.

# 4 A protogram interpreter

To experiment with the use of protograms, we have developed a protogram interpreter, dealing with the three special cases of interaction described in section 2. The interpreter is a "program skeleton", which accepts a collection of protograms, their modes of interaction and priorities, and handles the interaction between them.

More precisely, the currently implemented interpreter expects the following as description of the protograms:

- One procedure reading the input for all the protograms.
- For each protogram, a procedure computing its priority (as a function of the input).
- For each protogram, a procedure (henceforth "effect procedure") computing — as a function of the input, of this protogram's priority, and of any constraints imposed on it — the protogram's effects on the system's actions and the constraints it imposes on other protograms.
- For each protogram, an indication of the way it interacts with lower-priority protograms (by overriding, constraining, or compromise).
- One procedure running after all effect procedures, reading a common data structure that was modified by each of them, performing the resulting actions.

Even in its present, limited form, the interpreter is already suitable for a variety of applications. We discuss three of them, that were actually implemented with the interpreter, in the next section.

# 5 Three applications

The three applications described below deal with three very different tasks, but all requiring some kind of interaction between several conflicting goals; this makes them suitable for specification and implementation by means of protograms. The three examples also demonstrate the three different types of protogram interaction handled by the interpreter.

## 5.1 a sorting program for a list of employees

As discussed in section 1, the main motivation for the concept of protograms is from robotics applications. This method is especially suited for robotics, and for other systems involving continuous reaction to new input. However, protograms are sometimes also useful in data processing systems, applying an input-to-output transformation. One example of such a use was discussed in section 2, and was implemented using the protogram interpreter — a program sorting a list of employees.

The input list includes, for each employee, his name, his salary, and an indication whether he is the manager. The list is sorted alphabetically as far as possible (i.e. minimizing the number of out-of-order pairs) while putting

the name of the manager on top; if there is no manager, the list is simply sorted alphabetically.

The program is implemented by supplying the interpreter with two protograms, "place the manager at head of list" (the higher-priority one) and "sort alphabetically" (the lower-priority one), which interact by constraining.

## 5.2 a robot movement simulation program

Consider a robot with two protograms, "Get to point A quickly" and "Get to point A safely". Those two protograms control one linear parameter: the speed of the robot, determined by a compromise between them. Relative priorities can change dynamically, depending on road conditions and perhaps also depending on instructions from the robot's operator. To demonstrate this type of protogram interaction, we developed a simulator for such a robot.

The program reads a file describing the safety conditions along the path (in a real system, the robot will have to perceive the path and infer the safety conditions). The safety conditions determine, for each possible speed of the robot, the probability of crashing into an obstacle; this probability increases with speed.

The program uses the safety conditions it reads to determine the speed of the robot. The program has some default behaviour, "always move at the speed that gives you X chance of crashing,,. It also accepts interrupts from the user, who may instruct it to go faster or slower; such an instruction from the user will modify the behaviour of the robot from then on, until the next instruction.

Once the speed of the robot is determined, the program decides randomly, according to the appropriate probability, whether the robot has crashed into an obstacle (in a real system, the robot will try to move while avoiding perceived obstacles, and this task will be harder if it is moving fast). If it hasn't crashed, the program calculates how far the robot will go during the next cycle, reads ahead to the appropriate place in the input file, and repeats the same operation.

The program ends either when the robot crashes, or when the robot reaches the end of the path. In the latter case, the program reports how long it took the robot to get there. The program can be used as a game, in which the player can make his decisions regarding what instructions to give the program, the purpose being to get to the end as fast as possible without crashing.

The program is implemented with two protograms, using the protogram interpreter. The interaction is by compromise.

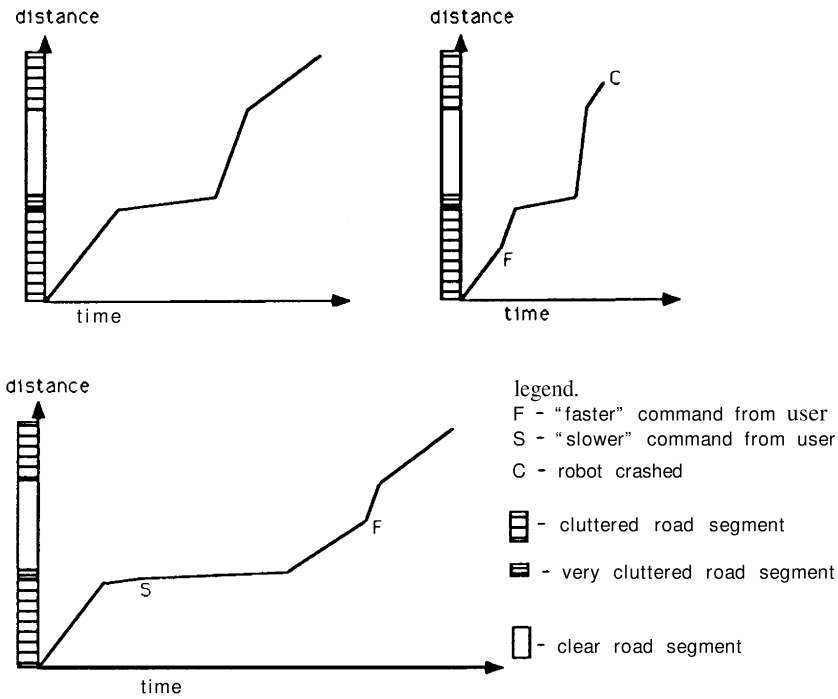


Figure 2: possible runs of robot movement simulator

Some possible runs of the program are illustrated in figure 2.

## 5.3 A "stop the moving object" program

This subsection describes an application which, so far, has been the main focus of the "protogram" project. It is the type of domain that is most suited for protograms, allows the use of state-of-the-art robotics equipment, and contains several interesting questions and opportunities for experimentation.

### 5.3.1 The task

The task is to observe several objects on a table, some of them in motion, and use an arm to stop any object that is in danger of falling. If more than one object appears to be in danger, the program must decide which object to stop.

The decision is made based on several criteria: how close the object is to the edge, how fast it is moving, how important it is to keep this object

from falling (depending e.g. on whether it is breakable), and how certain the perception of the object is[2].

Once the program decides to stop an object, it needs to move an arm into the object's path, and should decide exactly where to place it. This decision depends, of course, on the object's speed, which determines how much it will advance in the time it takes to move the arm. It also depends on the degree of certainty in its perception. If there is more uncertainty in the movement's speed, the arm should be placed further away from the object's present position, so that it doesn't undershoot the object. On the other hand, of there is more uncertainty in the movement's direction, the arm should be placed closer to the object's present position, so that it doesn't miss the object.

### 5.3.2 Implementation

So far, we have implemented a simulator of the system, using a real vision system but not using a real arm. This simulator, again, was implemented using the protogram interpreter.

The vision system used is a state-of-the-art system at Teleos Research, Inc., which produces, for each time frame, an optical flow matrix. The program analyzes each frame provided by the vision system, and tries to find a moving object.

Currently, our object-detection method can only find one object. The program reads two series of frames in parallel, each produced by the vision system perceiving one moving object (this can be seen as a simulation of looking at two tables side by side, each carrying one object, and having just one arm). The program decides at each instant which of the two objects is in danger, and if both are, which one to stop.

The task of stopping each of the two objects is a protogram; there is also a protogram aimed at keeping the arm free (so that an object only gets stopped if it is in some minimal degree of danger of falling). The interaction

[2] If we are dealing with actual vision equipment, there will always be some uncertainty in the perception, and some objects may be perceived more clearly than others. The proper way of taking this criterion into account is not clear-cut — if an object is perceived less certainly, with more noise, is it more urgent to stop it or less urgent? — but it clearly needs to have some effect. Our decision, currently, is that, when urgency based on the other criteria is above some threshold, uncertainty decreases the urgency (since we are not certain that the object is really in this much danger); when urgency based on the other criteria is below that threshold, uncertainty increases the urgency (since the object may be in more danger than we think).

between the protograms is by overriding, and relative priorities are dynamically determined according to the speed of each object, its distance from the edge, and the degree of uncertainty in its perception.

# 6 Related work

The modularity of the protogram framework makes it similar in appearance to the *blackboard architecture* [2, 3]. Indeed, as was mentioned in section 1, they are both motivated by complexity and distributivity of knowledge. However, protograms are different from blackboard systems in two related respects. First, unlike blackboard "knowledge sources," each protogram is associated with an ideal *behavior,* and with a measure of deviation from the ideal. Second, in a blackboard system at each round exactly one of the knowledge sources that triggers is selected to run; any interaction between knowledge sources is limited to information recorded in the blackboard, the common data base. Protograms, in contrast, are not a winner-takes-all construct; some or all of the active protograms can be taken into account in determining the actual behavior, so the protograms framework stresses interaction among protograms.

The protograms framework is also reminiscent of Brooks' *subsumption architecture* [6], in which simple bug-like behaviors are captured in hardware, and then aggregated hierarchically to yield increasingly more complex (though, to date, still bug-like) behaviors. This subsumption architecture can be expressed very naturally in the protogram framework. In fact, it is tempting to view protograms as a software version of Brooks' hardware, especially in light of the similar robotics applications. For several reasons, however, we discourage this view. First, Brooks' approach is associated with philosophical claims with which we disagree. Brooks has championed anti-representation, arguing that intelligent machine behavior is an emergent phenomenon which makes no use of symbolic representation [7]. We on the other hand intend both sensory-motor protograms and formal-symbolic ones. Second, as we discussed in section 2, we intend for quite flexible combinations of behaviors. Brooks' combination of behaviors appears to be more rigid: most combinations that have appeared in the literature have been a simple overriding, and all behaviors have had fixed priorities. Finally, the subsumption architecture is geared specifically towards robotics; indeed, it is discussed entirely in hardware terms. We, on the other hand, propose protograms as a general programming methodology. Although our primary applications are in robotics, we intend that protograms be used in other software tasks as well.

The idea of protograms was actually inspired by the literature, primar-

ily in psychology and linguistics, on the theory of abstraction and family *resemblance.* Protograms are, in essence, an attempt to abstractly specify programmed courses of action — i.e. specify the requirements for a course of action in a generalized way, so that some flexibility is possible and several such courses can be accommodated. Several other AI projects can also be seen as applications of the theory of abstraction to their domains; for example, number-to-symbol translation[8, 9, 10] is an application of abstraction to numerical values.

Specifically, *prototype theory* was the original source of the idea of protograms, and of the current terminology we use for describing it. Prototype theory holds that the basis of abstraction is not the notion of *set* but rather those of *prototype* and *similarity to a prototype.* The argument, which dates back at least to Wittgenstein [ 11], is that one never has necessary and sufficient conditions for determining that something is of a certain sort, but rather one has a prototype for that sort, and a measure of how close any object is to that prototype. Thus a dog may be a fairly prototypical animal, a kangaroo less typical, and a Volkswagen even less so. Attempts have been made to confirm these ideas in psychological experiments, most notably by E. Rosch and her associates (cf. [ 12]). A rather exhaustive account of family resemblance and related topics appears in Lakoff's [13]. An alternative theory of abstraction, which avoids some of the philosophical and psychological problems with prototype theory, on which further investigation of protograms can be based, is *measurement-omission theory,* which holds that the basis of abstraction is regarding certain attributes of an entity as quantitative, and abstracting away from the specific measurements to form a general' concept. The theory was first described in [14]. A rigorous discussion of the basic idea of measurement-omission, contrasting it with other theories of abstraction, can be found in [15].

# 7 Further research

Future research on the subject of protograms will include more detailed experimentation with applications of protograms, as well as more theoretical work on the connection to theories of abstraction. We intend to try building some applications actually controlling — rather than just simulating — a robot or an arm. We also intend to expand the interpreter to a more general implementation, and experiment with implementing more examples.

## 7.1 Possibilities for a more general interpreter

Let us now look at some more general possible implementations of a protogram interpreter, based on the mathematical definition of section 3.

We don't believe there is any pratical way to program an efficient and fully general arbiter for optimising $\sum_C(P * S_i(v_i))$. There is, however, one special case in which the optimisation can be performed easily and efficiently: when the possible $v_i$ values can be divided into a finite, relatively small set of ranges, such that each such range is indistinguishable to all protograms (i.e. for any $S_i$ function from any protogram, the function will return the same value on all this range). In this case, all the arbiter need do is compute the weighted sum for a representative value from each of the ranges, find the range which achieves the maximum sum, and then take any value from that range.

We submit that many applications do fit naturally into this case. In many applications, the $v_i$ values will be either numbers or vectors, and the $S_i$ functions can return a relatively small set of values (for example, whole numbers from 0 to 10). In this case, each protogram can represent each of its $S_i$ functions as an array of possible return values, and, for each one, a list of ranges for which this is the value; the arbiter can then find all the intersections of the ranges supplied by the different protograms, and find the optimal value as above. All the examples discussed so far, except for the array sorting program, fit into this pattern.

If we have a relatively small set of ranges, as above, but $V_i$ are arbitrary sets (so $v_i$ aren't necessarily numbers or vectors), then the arbiter can be implemented as follows:

We assume that ranges over $V_i$ are expressed in some language (which the arbiter need not understand). In addition to the protograms and the action function, the arbiter is given a "choice function", which accepts as arguments a set of ranges, performs their intersection, and returns either a value from that intersection or an indication that the intersection is null.

The arbiter can then list all combinations of ranges from different protograms, and computes the weighted sum for each one. It then finds the maximizing combination, and passes it to the choice function to get the value; if a null indication is returned, the arbiter tries the next best combination, until a value is returned.

This implementation is suitable, for example, for the array sorting application. There is one parameter, $V_1$, the set of possible permutations of the array. Ranges over $V_1$ may be expressed, for example, by conditions expressed in predicate calculus. The "put the manager on top" protogram will return one range — the condition "the manager appears in place 1".

The "sort alphabetically" protogram will return, as its ideal, the condition "the array is sorted" (or "the array has no out-of-order pairs"); as its second best range, the condition "the array has one out-of-order pair"; as its next best range, "the array has two out-of-order pairs"; etc. The choice function will have the knowledge necessary to perform efficient sorting (which it will perform the first time it is called). When intersecting a combination (of "the manager appears in place 1" with "the array has n out-of-order pairs"), it will check the sorted array to see whether the manager can be moved to the top while causing exactly n out-of-order pairs; if that is possible, it will return the resulting array as the value for vi.

## 7.2 Gofer robots and protograms

We intend to specifically try to use protograms in a robotics project now under development at J. C. Latombe's group at the robotics laboratory at Stanford — the Gofer project[l]. The project deals with controlling the operations of mobile robots in a building, with the goal of automating a variety of tasks such as delivery of mail to rooms, operation of machines, cleaning, etc. Below, we present a possible scenario for a Gofer robot, and discuss the various ways that protograms can be used in its implementation. For simplicity, we assume just one robot. In more complicated scenarios, we believe protograms may also be useful in allocating tasks among robots.

The scenario is as follows:

The robot needs to perform three tasks (in increasing order of urgency): vacuum-cleaning the floors in the rooms and corridors (a continuous task whenever it has nothing else to do), emptying trash cans (at a certain time each day), and delivering mail and coffee (whenever mail arrives at the building, and whenever anybody orders coffee). Each of those tasks can be a protogram, and the robot, whenever there is a conflict, needs to resolve them (in this case, by simple overriding).

Once a task has been chosen, protograms can be of further use in planning how to execute it, specifically in planning the robot's movements through the building.

When delivering (coffee or mail), if there are several items to deliver, the robot needs to plan an order of delivery. Once decided on the order, the robot's high-level behaviour would be: load as many items as you can carry, go to the room to which the first one is to be delivered, then to the second one, etc.; when you're empty-handed, return to the base (some room where the coffee machine sits and where the mail is delivered from outside), reload, and so on until deliveries are finished or until something new (mail

or an order for coffee) arrives (in which case the order of delivery should be recomputed).

In choosing the order of delivery, several considerations apply: coffee is more urgent than mail; higher-ranking people should be served first; the robot should try to minimize the distance it travels (so two people in the same room, or nearby rooms, should be served in one trip). Each of these considerations can be regarded as a protogram; the order of delivery is the parameter which the arbiter needs to choose, each protogram returns its ranges of orders according to its preferences, and the order is then chosen in the method described in my previous note.

In the task of vacuum-cleaning, the robot should move around the building, cleaning the room or corridor he's in, moving to another room or corridor, cleaning it, etc. until a more urgent task comes along. Protograms can be used in deciding, after you've finished cleaning a room or corridor, where to move next. Again, there are several considerations: it is better to move to a room or corridor which is nearby; it is better to move to a room or corridor which you haven't cleaned in a long while; and it is better to move to a room or corridor in which you can't see any people, so as not to bother anyone (the last is an example of dynamic priorities; if the robot chose a room to clean next, went to it and saw people in it, or if somebody came in before he finished cleaning, then the priority of cleaning this room goes down, which may cause the robot to immediately leave and choose another room). Here, the most natural way is to have a protogram for each room and corridor in the building, ordering "clean this place"; the interaction is by simple overriding, and the relative priorities are determined by the above considerations.

Also, in performing each of the tasks, once the robot has decided where it wants to go, it needs to perform the low-level task of getting there. The method described in the Gofer paper[ 1], sections 3.3-3.4, can be very naturally implemented with protograms to represent the "behaviour rules" and the artificial potential fields.

The above description assumes that protograms should be used at various levels of a system, while most of the system is specified conventionally. An alternative is a system of hierarchical protograms — a system specified entirely in the form of protograms, such that some higher-level protograms are themselves composed of lower-level protograms (e.g. the procedure returning the protogram's range and preferences, or the procedure returning the protogram's priority, could itself need to balance several considerations in reaching its answer, and could therefore be implemented as a system of protograms). Intuitively, this also looks like a promising concept, and we intend to investigate its possibilities.

# 8 Summary

In this paper, we described protograms — abstract courses of action, allowing for a range of specific actions, from which one is chosen by interaction with other protograms — as an approach to specifying and building complex systems dealing with multiple tasks. This approach provides flexibility, in being able to dynamically change relative priorities, and modularity, in being able to modify a system by adding new protograms to it. The general protogram interpreter, even in its present limited form, is useful for a variety of applications. Our planned future work includes building a more general interpreter, and using it in more applications, especially in actual robotics applications.

# References


[1] P. Caloud W. Choi J.-C. Latombe C. Le Pape and M. Yim. Indoor automation with many mobile robots. In *Proceedings IEEE International Workshop on Intelligent Robots and Systems,* Tsuchiura, Japan, 1990.

[2] B. Hayes-Roth. Blackboard architecture for control. *Journal of Artificial Intelligence,* 26:251–321, 1985.

[3] H. P. Nii. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine,* 7:38–53, 1986.

[4] 0. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *International Journal of Robotic Research,* 5( 1):90–98, 1986.

[5] J.C. Latombe. *Robot Motion Planning.* Klewer Academic Publishers, 1990 (to appear).

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

[7] R. A. Brooks. Intelligence without representation. In *Proc. of the Workshop on Foundations of Artificial Intelligence,* 1987.

[8] E. Davis. Solutions to a paradox of perception with limited acuity. In *Proc. of the First International Conference on Principles of Knowledge Representation and Reasoning,* pages 79-82, 1989.

[9] R. Parikh. The problem of vague predicates. In *Language, Logic, and Method,* pages 241-261. Reidel Publishers, 1983.

[10] N. Goyal and Y. Shoham. Numerical ranges to discrete symbols. manuscript, Robotics Laboratory, Stanford University.

[11] L. Wittgenstein. *Philosophical Investigations.* Basil, Blackwell & Mott, 1958.

[12] E. Rosch and C. B. Mervis. Family resemblences: Studies in the internal structure of categories. *Cognitive Psychology,* 7:573–605, 1975.

[13] G. Lakoff. *Women, Fire, and Dangerous Things.* University of Chicago Press, 1987.

[14] A. Rand. *Introduction to Objectivist Epistemology.* New American Library, 1979.

[15] D. Kelley. A theory of abstraction. *Cognition and Brain Theory, 7:329-357,* 1984.