Programmable Smart Membranes: Using Genetic Programming to Evolve Scalable Distributed Controllers for a Novel Self-Reconfigurable Modular Robotic Application

Forrest H Bennett III, Brad Dolin, and Eleanor G. Rieffel

FX Palo Alto Laboratory, Inc.
3400 Hillview Avenue, Bldg. 4
Palo Alto, CA 94304
{bennett, dolin, rieffel}@pal.xerox.com

Abstract. Self-reconfigurable modular robotics represents a new approach to robotic hardware, in which the "robot" is composed of many simple, identical interacting modules. We propose a novel application of modular robotics: the programmable smart membrane, a device capable of actively filtering objects based on numerous measurable attributes. Creating control software for modular robotic tasks like the smart membrane is one of the central challenges to realizing their potential advantages. We use genetic programming to evolve distributed control software for a 2-dimensional smart membrane capable of distinguishing objects based on color. The evolved controllers exhibit scalability to a large number of modules and robustness to the initial configurations of the robotic filter and the particles.

1 Introduction

A self-reconfigurable modular robot – fundamentally distinct from traditional robots – is composed of many simple, identical modules. Each module has its own power, computation, memory, motors, and sensors. The modules can attach to and detach from each other. Individual modules on their own can do little, but the robot, using the capabilities of the individual modules, can reconfigure itself to perform different tasks as needed.

Advantages of self-reconfigurable modular robots include physical adaptability to varying tasks and environments, robustness (since identical modules can replace each other in the event of failure), and economies of scale in the manufacturing process. Physical implementations of modular robots include CMU's I-cubes [23], USC-ISI's Spider Link Models [5], Dartmouth's Molecular Robots [13] and Crystalline Robots [21], MSU's Reconfigurable Adaptable Micro-Robot [22], Johns Hopkins University's Metamorphic Robots [19], as well as Xerox PARC's PolyBot [26], and Proteo robots [2].

We propose a novel application of modular robotics which fully exploits such advantages: the smart membrane. This device is an active, selective filter which can in principle distinguish objects based not only on size, but also on shape, color, angle of movement, and indeed any property which can be measured with small sensors, or computed with such information. The applications of a physical implementations of a smart membrane are determined by the scale on which it is built: At the macro scale, a smart membrane could be used for parts sorting, while at the nanometer scale, such a device could be useful for purifying substances and augmenting biochemical processes.

The problem of developing effective software for modular robotic applications like the smart membrane is recognized as one of the central challenges to the development of practical self-reconfigurable modular robots. The ultimate aim is to have the software completely decentralized and completely autonomous, so that tasks can be performed without reference to a central controller. Decentralized control takes advantage of the computational power of the individual modules and requires less communication bandwidth. All modules run the same program, but behave differently depending on individual sensor values, internal state, and messages received from nearby modules. The challenge is to design software that is robust to initial conditions, scales well as the number of modules increases, and acts at the local level but achieves useful global behavior.

We use genetic programming to evolve robust, scalable, distributed controllers for a simulated, 2-dimensional smart membrane capable of sorting particles based on color. The smart membrane's dimensions are flexible, as we discuss below, but a typical instantiation is around sixty modules across, and eight modules high. An arbitrary number of particles, the size of single modules and colored either blue or red, are placed above the artificial membrane. The modular robot then actively filters the particles – by pushing and pulling them, and creating temporary transport channels – so that the final configuration of the simulated world has all the red particles on top of the smart membrane, and all the blue particles below, with the membrane structure remaining reasonably intact.

2 Related Work

The smart membrane represents a novel application of modular robotics, and indeed the idea of a programmable filter is original for any implementation. Keller and Ferrari [11] provide a micro scale passive filter with sufficiently small holes to permit the passage of small desired biomolecules while at the same time preventing the passage of all larger molecules such as antibodies. A micromachined particle sorter was implemented by Koch et al. [12]. A nanometer scale active membrane was constructed [10] where the rate of water permeation through the membrane was controlled by pH levels and ionic strength. Drexler [6] gives a design for a unidirectional active molecule sorter capable of transporting specific molecules across a barrier. Drexler's sorter design was used as a component in the design of an artificial mechanical red blood cell [7], [8]. A membrane with porosity controlled by electroactive polymer actuators is described by Otero [18]. The smart membrane is unique in that its behavior can be determined by software, allowing for exceptional functional generality.

A small amount of centralized control software for modular robots has been developed. Casal and Yim [4] use a central controller for finding global strategies to achieve desired configurations. Global automated planning algorithms have been used for reconfiguring Crystalline Robots [21], [13]. Simulated annealing has been used to find near-optimal global methods for reconfiguring modular robots [20].

Almost no work has been done on developing decentralized control software for modular robots. Nearly all of such research has focused on hand-coding local rules for modular robotic tasks. Bonabeau, et al. [3] describe work in which local rules for reconfiguration of modular robots were hand-coded. Bojinov, et al. [2] hand-coded local control algorithms for a self-assembling robot made of rhombic dodecahedrons for two types of problems: reconfiguring in response to weight, and reconfiguring to grasp an object of unknown size and shape. Yim, et al. [25] hand-coded a local algorithm to enable the same kind of robot to reconfigure into any specified goal configuration. Kubica et al. hand-coded complex behaviors using local rules for robots which move by modular expansion and contraction [16], [17].

Automatically generating the modular robot control software is novel. ? [1] use genetic programming to evolve the control software for some simple modular robotic tasks. The tasks include locomotion of a group of modules through a crooked passage, cooperative propulsion of modules over a “slippery” bridge, and locomotion of a group of modules toward a goal. Their results exhibit robustness against module failures, changing environmental conditions, unknown environmental details, and varying initial conditions of the modular robots themselves.

3 Modular Robot Simulator

For our experiments, we wrote a simulator for the self-reconfigurable modular robot type designed and built by Pamecha, et al. at Johns Hopkins University [19], Figure 1a. We add sensing, communication, and processing capabilities that were not implemented in the modules built by Pamecha, et al.

Each square robot module occupies one grid location in the world. The state of a module includes its location, the most recent messages received from adjacent modules in each of eight directions, the values of its sensors, the values in its four memory locations, and its facing direction.

Directions are encoded as real values in [0.0, 1.0), where direction 0.0 is the robot module’s positive x axis, and direction values increase going around counter clockwise. The direction values used by each robot module are local to that module’s own frame of reference. The direction that a robot module is facing in the world is always direction 0.0 in the robot module’s frame of reference. Direction 0.25 is 90 degrees to the left of where it is facing in the world, etc.

3.1 Movement

Robot modules are able to move exactly one grid location in one of four directions: east, north, west, and south. A robot module cannot move by itself; it can only move by sliding against an adjacent robot module.

To slide, a robot module can push itself against another robot module that is adjacent to it at 90 degrees from the direction of motion (Figure 1b). Similarly, a robot module can pull against another robot module that is diagonally adjacent to it in the direction of motion. The pulling style of move is demonstrated in Figure 1b by considering step 2 as the initial configuration, and step 1 as the result of the move.

Robot modules can initiate two different types of moves to reconfigure the robot. A "single" move is the movement of a single robot module, and it succeeds if and only if that robot module is moving into an empty grid location and there is a module to push or pull against. A "line" move is the movement of an entire line of robot modules, and it succeeds if and only if the front-most module in the line is moving into an empty grid location, and there is a module to push or pull against. A line move can be initiated by any robot module in a line of robot modules.

Fig. 1. (*a*) Photograph of the physical hardware for two self-reconfigurable modules [19]. (*b*) Diagram of the basic single move operation for the style of self-reconfigurable robot module used for the smart membrane. Before the move, the two modules share a connected edge (step 1); afterwards two modules are still connected at a shared corner (step 2). Reprinted by permission © ASME

whether the object in the adjacent grid location in the current facing direction is red or blue. The eight directions for sensors are: 0.0 = east, 0.125 = northeast, 0.25 = north, 0.375 = northwest, 0.5= west, 0.625 = southwest, 0.75 = south, and 0.875 = southeast. A module's sensor readings are in units of intensity in [0.0, 1.0], where the intensity is the inverse of the distance to the thing sensed; zero means that the thing was not sensed at all, and one means that the thing was sensed in the immediately adjacent grid location.

4. The Smart Membrane Problem

In the training version of the problem, a single module-sized red or blue square, the "particle," is placed directly above the smart membrane. The membrane is required to reconfigure itself – pushing and pulling the particle, and creating temporary transport channels – so that the particle remain above the membrane, if red, or is moved to the bottom, if blue. In addition, the membrane itself must end up close to its initial position, though the individual modules within the membrane do not need to end up close to their original positions. After training, this behavior should generalize over a wide range of membrane sizes and particle positions.

4.1 Function Set

The following basic function set was used to evolve the control software for the smart membrane:

- `(MoveSingle direction)` causes the current robot module to move in the relative direction `direction` if it can move. This function returns true if the module is able to move, and false otherwise.
- `(MoveLine direction)` causes an entire line of robot modules to move in the relative direction `direction` if they can. The line of modules to move is the connected line of modules collinear with the current module in the relative direction `direction` of the current module. This function returns true if the line of modules is able to move, and false otherwise.
- `(Rotate direction)` rotates the robot module by the amount `direction`. This does not physically move the module, it merely resets the internal state of the robot module's internal facing direction. This function returns the value of `direction`.
- `(ReadMessage direction)` reads the real-valued message from the adjacent module (if any) that is location at the relative direction `direction` to the receiving module. The `direction` is interpreted mod 1, and then rounded to the nearest eighth to indicate one of the eight adjacent grid locations. This function returns the value of the message read.
- `(SendMessage message direction)` sends the real-valued message `message` from the sending module to an adjacent module (if any) in the direction `direction`, relative to the frame of reference of the sending module. The `direction` is interpreted as in `ReadMessage`. This function returns the value of the message sent.
- `(ReadSensorSelf direction)`, `(ReadSensorWall direction)`, `(ReadSensorParticle direction)` read the intensity value of the sensor for detecting another module, a wall, or a particle, respectively. Intensity is the inverse of the distance to the closest object of interest at the relative direction `direction`. The intensity is zero if there is no object of interest in that direction. This function returns the intensity value.
- `(IsBlue direction)`, `(IsRed direction)` returns true if there is a blue or red particle, respectively, immediately next to this module in direction `direction`, and false otherwise.
- `(GetMemory index)` gets the current value of the robot module's memory numbered `index`. This function returns the value of the memory.
- `(SetMemory index value)` sets the value of the robot module's memory numbered `index` to `value`. This function returns `value`.
- `(ProgN arg1 arg2)` evaluates both `arg1` and `arg2`, and returns `arg1`. This function allows for sequential command execution.
- `(ProtectedDivide num denom)` divides `num` by `denom`, returning the result, or 1 if the operation results in an error or an infinite value.
- `(ProtectedModulus num denom)` calculates `num` modulus `denom`, returning the result, or 1 if the operation results in an error or an infinite value.
- `(And arg1 arg2)`, `(If test arg1 arg2)`, `(Less arg1 arg2)`, with the usual definitions [15].

4.2 Terminal Set

The terminal set used to evolve the control software for the smart membrane includes only `(GetTurn)` and some numerical constants. `(GetTurn)` returns the current value of the "turn" variable, which is set to zero at the beginning of the simulation and is incremented each time all of the robot's modules are executed.

Six constant-valued terminals `0.0`, `0.25`, `0.5`, `0.75`, `1.0`, and `–1.0` are given. Program trees can use arithmetic to create other numerical values.

4.3 Fitness

Though the final smart membrane distributed control software generalizes to larger simulation worlds, the world used for fitness evaluation is a square with 10 units on a side. (The perimeter grid locations of the entire world contain wall boundary objects through which the robot modules cannot pass.)

The fitness for an individual program is calculated as a weighted average of the fitness in 24 simulation worlds. Each simulation world is initialized with the smart membrane's robot modules arranged in a vertically centered rectangle extending 8 units from the left wall to the right wall. In twelve of the worlds, the robotic smart membrane is of height three modules; in the other twelve, the membrane is of height four modules. At the beginning of each fitness evaluation, the values of the communication buffers and the values in all the robot's memory locations are initialized to zero. The facing direction of the modules is determined randomly for each world.

The fitness is computed by running a simulation of the robot's actions in each world. The simulation is run for 10 execution "turns." In each turn, the evolved program tree is executed once for each module. The order in which the modules' programs are executed is also determined randomly for each world, though the order remains constant over the 10 execution turns.

Each fitness world includes a single particle to be filtered, which is placed on top of the membrane. Particles are colored either blue or red. The membrane must filter the particles so that red particles remain above the filter, while blue particles are transported through the filter to end up below the membrane. An evenly distributed sample of possible horizontal positions for the particle is contained in the 24 fitness worlds. In six of the worlds, the particles that appears initially above the membrane is colored red, in which case the particles does not need to be moved at all. In the other 18 worlds, the particles that appears initially above the membrane is colored blue, so the membrane must transport the particles through the membrane to the other side.

The fitness is a weighted sum of two measurements, averaged over all turns: the location of the particle relative to the membrane, and the current absolute position of the membrane modules. The former term is given by the average offset in the vertical dimension of the modules from the particle. This value is normalized so that a score of 1 means that the membrane made no progress in moving the particle toward its correct final position, whereas a score of 0 means that the particle is either above or below the membrane, as appropriate. This term has the effect of making sure that each object is moved toward the appropriate final relative position.

The latter term is given by the average absolute displacement of each module from the horizontal central axis of the starting membrane position. This term penalizes the robot for failing to keep the original structure and position of the membrane intact, while the object is being filtered.

It was determined empirically that an effective setting of these weights was:

$$95 * (\textit{particle displacement}) + 5 * (\textit{membrane distance from center}) \,. \qquad (\mathbf{0})$$

This setting is fairly arbitrary, however, and other weights seemed to work as well. Note that the complexity of the task – and the difficulty of achieving a useful fitness gradient in initial generations – forced us to adopt this rather ad hoc fitness measure.

4.4 Parameters

The population size for this problem is 72,000 individuals. The crossover and mutation rates are 99%, and 1% respectively. Crossover and mutation select nodes with differential probability, depending on position within the tree. Leaf nodes have a 10% chance of being chosen for crossover, whereas internal nodes have an 89% chance. Leaf nodes have a 0.5% chance of mutation, as do internal nodes. We use the generational breeding model with tournament selection, and a tournament size of 7. Elitism is used, which insures that the most fit individual in each generation is cloned into the next generation. The method for creating program trees in generation 0 and in mutation operations is the "full" method [14]. The maximum depth for program trees in generation 0 is 5. The maximum depth for program trees created by crossover and mutation is 9. The maximum depth of subtrees created by mutation is 4. We use strongly typed genetic programming [9].

The population is divided into semi-isolated subpopulations called demes following Wright [24]. Breeding is panmictic within each deme, and rare between demes. The parallelization scheme is the distributed asynchronous island approach [15]. The communication topology between the demes is a toroidal grid with periodic boundary conditions. Each processor communicates only with its four nearest neighbors. At the end of each generation, 2% of the population is selected at random to migrate in each of the four cardinal directions.

Fig. 2. This series shows the behavior of the evolved smart membrane control software on one test case. The leftmost scene shows the initial condition for the simulation, the middle scene shows an intermediate time step, and the rightmost scene shows the final

5 Results

A working solution to this task emerged in generation 32 of the run. The solution was composed of 49 primitive function references and 32 real valued constants. The behavior of this solution is shown in Figure 2 for a single initial world condition at three points in time during the simulation.

Although this program was evolved in a 10x10 world, with membrane heights of 3 and 4, and one object filtered at a time, the same program successfully controls much larger membranes which can filter multiple objects at the same time. For example, the program functioned successfully when installed on each of the 496 robot modules in a 64x64 world, with a membrane height of eight modules and six objects to be filtered (three red and three blue) spaced evenly along the top of the membrane. An example of this out-of-sample generalization is shown in Figure 3. The same program works just as well with intermediate numbers of modules and particles, and with different particle placement (so long as there is at least several modules of space between particles).

6 Conclusions and Future Work

We have demonstrated an effective means of evolving distributed control software for the smart membrane, a difficult modular robotic task. Furthermore, we have demonstrated robustness of the evolved program to particle placement, and scalability as the number of robot modules and particles increases.

Acknowledgements

Fig. 3. This shows the final configuration of the evolved smart membrane control software on an out-of-sample world where the world size, depth of the membrane, width of the membrane, and number of foreign object to be sorted are all different from any of the worlds used in training. (The top and bottom of the world, devoid of modules, are not shown.)

up the computing cluster and for conversations about modular robots, Zoe Abrams for her programming work and technical suggestions, Jeremy Kubica for his work in hand-coding some modular robotic algorithms and so helping us to debug the evolutionary technique, and Adil Qureshi for his Gpsys 1.1 java genetic programming source code on which our system is based.

References

1. Bennett III, F.H, Rieffel, E.G.: Design of Decentralized Controllers for Self-Reconfigurable Modular Robots Using Genetic Programming. Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware (2000) 43-52
2. Bojinov, H., Casal, A., Hogg, T.: Emergent Structures in Modular Self-Reconfigurable Robots. IEEE Intl. Conf. on Robotics and Automation (2000) 1734-1741
3. Bonabeau, E., Dorigo, M., Theraulaz, G.: Swarm Intelligence: from Natural to Artificial Systems. Oxford Univ. Press (1999)
4. Casal, A., Yim, M.: Self-Reconfiguration Planning for a Class of Modular Robots. Proceedings of SPIE'99, Vol. 3839 (1999) 246-256
5. Castano, A., Chokkalingam, R., Will, P.: Autonomous and Self-sufficient Conro Modules for Reconfigurable Robots. Proceedings of the Fifth International Symposium on Distributed Autonomous Robotic Systems (DARS) (2000)
6. Drexler, K.E.: Nanosystems: Molecular Machinery, Manufacturing, and Computation. John Wiley & Sons, Inc. (1992)

7. Freitas Jr., R.A.: Respirocytes: High Performance Artificial Nanotechnology Red Blood Cells. NanoTechnology Magazine 2 (Oct. 1996) 1, 8-13 (8)
8. Freitas Jr., R.A.: Nanomedicine, Vol. 1: Basic Capabilities. Landes Bioscience (1999)
9. Haynes, T., Wainwright, R., Sen, S., Schoenfeld, D.: Strongly Typed Genetic Programming in Evolving Cooperation Strategies. Proceedings of the Sixth International Conference on Genetic Algorithms. Morgan Kaufmann (1995) 271-278
10. Ito, Y.: Signal-responsive Gating by a Polyelectrolyte Pelage on a Nanoporous Membrane. Nanotechnology, Vol. 9 No. 3, (Sep. 1998) 205-207
11. Keller, C.B., Ferrari, M.: Microfabricated Capsules for Immunological Isolation of Cell Transplants. US Patent No. 5,893,974 (13 Apr. 1999)
12. Koch, M., Schabmueller, C., Evans, A.G.R., Brunnschweiler, A.: A Micromachined Particle Sorter: Principle and Technology. Tech. Digest, Eurosensors XII, Southampton, UK, September 13-16 (1998)
13. Kotay, K., Rus, D., Vona, M., McGray, C.: The Self-Reconfiguring Robotic Molecule: Design and Control Algorithms. Proceeding of the 1998 International Conference on Intelligent Robots and Systems (1998)
14. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
15. Koza, J.R., Bennett III, F.H, Andre, D., Keane, M.A.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufmann (1999)
16. Kubica, J., Casal, A., Hogg, T.: Agent-Based Control for Object Manipulation with Modular Self-reconfigurable Robots. Submitted to Intl. Joint Conf. on AI (2001)
17. Kubica, J., Casal, A., Hogg, T.: Complex Behaviors from Local Rules in Modular Self-Reconfigurable Robots. Submitted to IEEE ICRA (2001)
18. Otero, T.F.: EAP as Multifunctional and Biomimetic Materials. Smart Structures and Materials 1999: Electroactive Polymer Actuators and Devices 26-34
19. Pamecha, A., Chiang, C.J., Stein, D., Chirikjian, G.S.: Design and Implementation of Metamorphic Robots. Proceedings 1996 ASME Design Engineering Technical Conference and Computers and Engineering Conference (1996) 1-10
20. Pamecha, A., Ebert-Uphoff, I., Chirikjian, G.S.: Useful Metrics for Modular Robot Motion Planning, IEEE Transactions on Robots and Automation, Vol.13, No.4 (Aug. 1997) 531-545
21. Rus, D., Vona, M.: Self-Reconfiguration Planning with Compressible Unit Modules. IEEE Int. Conference on Robotics and Automation (1999)
22. Tummala, R.L., Mukherjee, R., Aslam, D., Xi, N., Mahadevan, S., Weng, J.: Reconfigurable Adaptable Micro-Robot. Proc. 1999 IEEE International Conference on Systems, Man, and Cybernetics (Oct. 1999)
23. Ünsal, C., Kiliççöte, H., Khosla, P.K.: A 3-D Modular Self-Reconfigurable Bipartite Robotic System: Implementation and Motion Planning. Submitted to Autonomous Robots Journal, special issue on Modular Reconfigurable Robots (2000)
24. Wright, S.: Isolation by Distance. Genetics 28 (1943)114-138.
25. Yim, M., Lamping, J., Mao, E., Chase, J.G.: Rhombic Dodecahedron Shape for Self-Assembling Robots. Xerox PARC SPL TechReport P9710777 (1997)
26. Yim, M., Duff, D.G., Roufas, K.D.: PolyBot: a Modular Reconfigurable Robot. IEEE Intl. Conf. On Robotics and Automation (ICRA) (2000)