Design of Decentralized Controllers for Self-Reconfigurable Modular Robots Using Genetic Programming

Forrest H Bennett III
FX Palo Alto Laboratory, Inc.
3400 Hillview Avenue, Bldg. 4
Palo Alto, CA 94304
bennett@pal.xerox.com

Eleanor G. Rieffel
FX Palo Alto Laboratory, Inc.
3400 Hillview Avenue, Bldg. 4
Palo Alto, CA 94304
rieffel@pal.xerox.com

Abstract

Advantages of self-reconfigurable modular robots over conventional robots include physical adaptability, robustness in the presence of failures, and economies of scale. Creating control software for modular robots is one of the central challenges to realizing their potential advantages. Modular robots differ enough from traditional robots that new techniques must be found to create software to control them. The novel difficulties are due to the fact that modular robots are ideally controlled in a decentralized manner, dynamically change their connectivity topology, may contain hundreds or thousands of modules, and are expected to perform tasks properly even when some modules fail. We demonstrate the use of genetic programming to automatically create distributed controllers for self-reconfigurable modular robots.

1 Introduction

Modular hardware that can reconfigure itself is usually referred to as a self-reconfigurable modular robot however little it resembles the common view of what a robot is. A self-reconfigurable modular robot is composed of many 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 [Ünsal 2000a, Ünsal 2000b], USC-ISI's Spider Link Models [CONRO], Dartmouth's Molecular Robots [Kotay 1998] and Crystalline Robots [Rus 1999], MSU's Reconfigurable Adaptable Micro-Robot [Tummala 1999], John's Hopkin's University's Metamorphic Robots [Pamecha 1996], as well as Xerox PARC's PolyBot [Yim 2000], and Proteo robots [Bojinov 2000].

The problem of developing effective software for modular robots 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, whether human or machine. The hope is that eventually hundreds, or even millions, of modules will work together. 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 acts at the local level but achieves useful global behavior.

Our approach uses evolutionary techniques to generate such software. This approach is particularly general and adaptive due to mechanisms analogous to variation, selection, and retention in biology. There are several advantages of evolutionary techniques over current methods. To begin with, our approach generates the software automatically. Secondly, it is a general method that can generate software for different task types. Thirdly, it can design decentralized controllers that are robust against module failures, changing environmental conditions, unknown details in the environment, and varying initial conditions of the modular robots themselves.

2 Related work

Almost no work has been done on developing decentralized control software for modular robots. Bojinov, et al. [Bojinov 2000] at Xerox PARC 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.

[Yim 1997] at Xerox PARC hand-coded a local algorithm to enable the same kind of robot to reconfigure into any specified goal configuration. Bonabeau, et al. [Bonabeau 1999] describe work in which local rules for reconfiguration of other types of modular robots were hand-coded.

A small amount of centralized control software has been developed. Casal and Yim [Casal 1999] at Xerox PARC use a central controller for finding global strategies to achieve a desired configuration. Researchers at Dartmouth Robotics Lab [Rus 1999], [Kotay 1998] have a global automated planning algorithm for their Crystalline Robots to move from one configuration to another. Pamecha et al. [Pamecha 1997] of Johns Hopkins University, Robot Kinematics and Motion Planning Lab used simulated annealing to find "near-optimal" global methods for getting their "metamorphic robots" from one configuration to another.

Chen [Chen 1996] of Nanyang Technological University, Singapore and Caltech has used genetic algorithms and other evolutionary techniques to find algorithms that enable a robot to reconfigure into a shape suited to various specific tasks. While he calls his robots "modular robots," they are not modular robots in the sense used in this paper because they are not capable of self-reconfiguring, they are not homogeneous (i.e. the modules are not identical), and the modules do not have their own computing and power resources.

Yamasaki, K., et al., [Yamasaki 1998] have used genetic programming to design local rules by which a line of rhombuses, with joints at the corners, can change into a specified shape. Simulation was done for a module line of ten rhombuses. Control programs are separately evolved for each goal configuration. This work differs from our work in two ways. It does not apply to the type of self-reconfiguring modular robots discussed in this paper because the topological connectivity of the structure being controlled does not change, and this work used open loop control whereas our work uses closed loop control.

3 Overview

We used genetic programming to design distributed controllers for a one type of self-reconfigurable modular robot. We describe the encoding that we use in our approach, and then describe six different experiments in which genetic programming evolved a distributed controller to solve a problem, including problems involving module failures, random obstacles, and varying initial conditions. We also describe a set of studies we did for one of the problems.

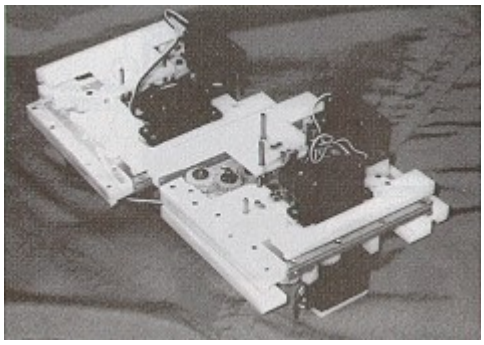

Figure 1: Photograph of the physical hardware for two self-reconfigurable modules [Pamecha 1996]. Reprinted by permission © ASME.

4 Experiments

For our experiments, we wrote a simulator for the self-reconfigurable modular robots designed and built by Pamecha, et al. at Johns Hopkins University [Pamecha 1996]. The square modules move in a two dimensional plane by sliding against each other (Figure 1). A module cannot move at all unless it is connected to another module that it can push or pull against. Each module occupies one grid unit in the simulated world, and all moves are in whole grid units. We add sensing, communication, and processing capabilities that were not implemented in the modules built by Pamecha, et al.

4.1 The robot modules

Each square robot module occupies one grid location in the world. The state of a module includes its location, the values of 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, whether or not the module is broken, and its facing direction.

Directions: 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.

Sensors: Each module has eight sensors for detecting other robot modules, and eight sensors for detecting walls and obstacles. 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 intens-

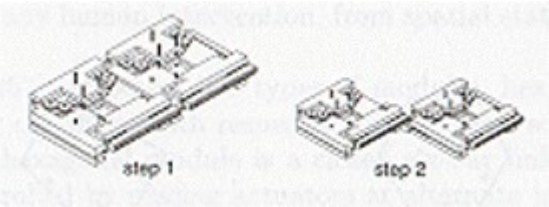

Figure 2: Diagram of the basic `MoveSingle` operation for the self-reconfigurable robot modules used for the experiments in this paper. Before the `MoveSingle`, the two modules share a connected edge (step 1), and then after the MoveSingle the two modules are still connected at a shared corner (step 2). Reprinted by permission © `ASME`.

ity 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.

Movement: Robot modules are able to move only in the 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 as demonstrated in Figure 2. 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 2 by considering step 2 as the initial configuration, and considering 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.

4.2 The problems

We solved six problems to demonstrate how our uniform approach can be easily targeted to produce distributed controllers that achieve different objectives.

For each problem, we use 9 robot modules arranged in a 3x3 square at the beginning of each fitness evaluation. The robot is simulated in a square world 40 units on a side. A solid wall of impenetrable obstacles surrounds the entire world (Figures 3 – 6).

Passage problem: The robot must find the narrow, winding passage (Figure 4) that divides the world and travel through this passage as quickly as possible. The robot is initially located at the bottom center of the world. The order in which the modules are placed and the initial facing direction for each module are determined randomly once at the beginning of the run, and held constant for all fitness evaluations.

Switchback problem: The robot must find the narrow switchback passage (Figure 3) that divides the world, and navigate through this passage as quickly as possible. Solving this problem requires avoiding getting stuck in the local optimum inside of the passage. For each fitness evaluation, the modules are placed in numerical order and the initial facing directions are set randomly.

Passage and carry broken module problem: The robot begins with one broken module in its configuration. It must carry this broken module as it accomplishes the task of finding and passing through the narrow passage (Figure 4) as quickly as possible. The modules have a sensor to detect broken modules. The broken modules cannot move or send or receive messages. The initial placement of the modules is the same as in the switchback problem.

Passage and eject broken module problem: The robot begins with one broken module in its configuration. It must eject this module out of the configuration as soon as possible, and then find and pass through the narrow passage (Figure 4) as quickly as possible. The modules have a sensor to detect broken modules. The broken modules cannot move or send or receive messages. The initial placement of the modules is the same as in the switchback problem.

Bridge problem: Modules inside a rectangular area cannot initiate moves so they must be moved by other modules outside the rectangle. The rectangle is seven rows high and two columns wide and can be thought of as a gap in the terrain that the robot can only get across by building a bridge out of its modules, and pushing or pulling those modules from either side of the gap. The robot must find the gap (Figure 5) and get across it to the top of the world. The robot is initially located at the center of the world configured in a 3x3 square. For each fitness evaluation, the modules are placed in numerical order and initially face east.

Move to goal problem: The robot must move from the center of each of 36 worlds, around randomly placed obstacles, to a goal which is placed at a different location in each world. Figure 6 shows one of the 36 worlds used for this problem. The robot modules can detect the goal direction. The modules are initially configured in a 3x3 square at the center of the world. In each of the worlds, the modules are placed in numerical order and initially face east.

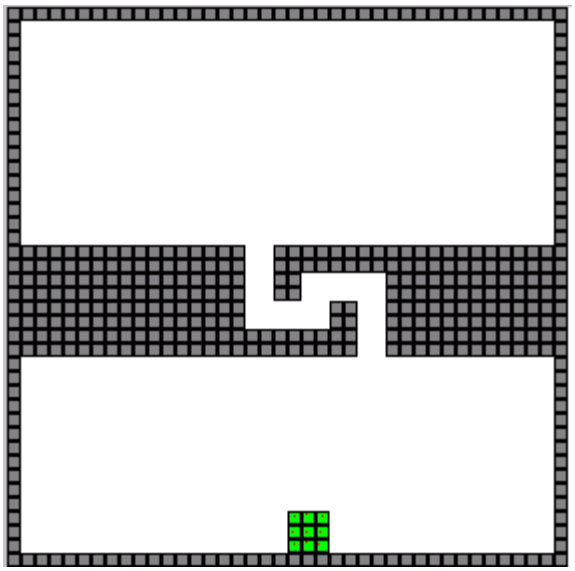

Figure 3: Diagram of the world for the switchback problem showing the switchback passage and the initial location of the robot modules at the bottom of the world.

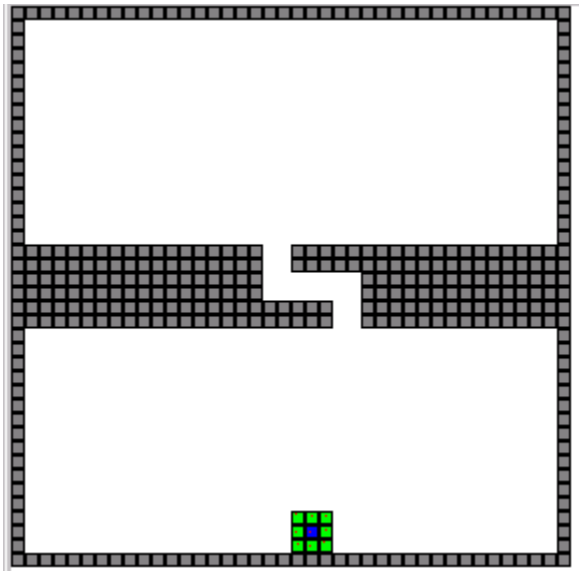

Figure 4: Diagram of the world for the passage-and eject-broken-module problem and the passage-and-carry-broken-module problem showing the narrow passage and the initial location of the robot modules at the bottom of the world, including the darker broken module in the center of the robot.

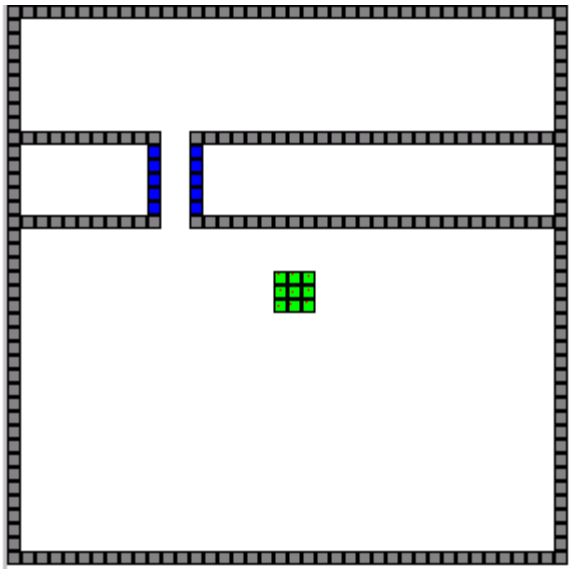

Figure 5: Diagram of the world for the bridge problem showing the narrow passage containing the seven by two rectangle where the robot modules cannot initiate moves but must be moved by other modules outside the rectangle. The modules must get across the rectangle and move to the top of the world.

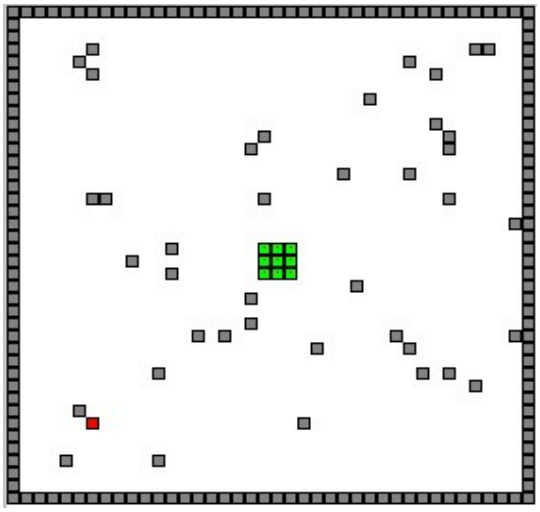

Figure 6: Diagram of the world for the move-to-goal problem showing one of the 36 worlds with randomly placed obstacles, a darker goal in the lower left corner of the world, and the initial location of the robot modules in the center of the world.

4.3 Function set

The following basic function set is used in all six problems presented in this paper.

BASIC_FUNCTION_SET = {MoveSingle, MoveLine, Rotate, ReadMessage, SendMessage, ReadSensorSelf, ReadSensorWall, GetMemory, SetMemory, And, If, Less, Divide, Modulus}

Two of the problems use one additional function. The passage-and-eject-broken-module problem and the passage-and-carry-broken-module problem add to the basic function set a function called readSensorSelfBroken.

Each of these functions are explained below.

- (And operand1 operand2) evaluates both operands and returns true if and only if both of the operands are true, and returns false otherwise.
- (Divide operand1 operand2) evaluates both operands, and returns operand1 divided by operand2. If operand2 is zero, this function returns MAX_FLOAT.
- (GetMemory index) gets the current value of the robot module's memory numbered index. This function returns the value of the memory.
- (If condition expression1 expression2) evaluates the Boolean condition. If condition is true, then expression1 is evaluated and its value is returned. If condition is false, then expression2 is evaluated and its value is returned.
- (Less operand1 operand2) evaluates both operands and returns true if operand1 is less than operand2, and returns false otherwise.
- (Modulus operand1 operand2) evaluates both operands, and returns the real-valued remainder of operand1 divided by operand2. If operand2 is zero, then this function returns MAX_FLOAT.
- (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. The line of modules can move if and only if the front-most module in the line is not blocked by an immovable wall or obstacle and there is another module adjacent to the line that it can push or pull against. This function returns true if the line of modules is able to move, and false otherwise.
- (MoveSingle direction) causes the current robot module to move in the relative direction direction if it can move. The module can move if and only if it is not blocked by any other object, and there is another robot module adjacent to it that it can push or pull against. This function returns true if the line of modules is able to move, and false otherwise.
- (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.
- (ReadSensorSelf direction) reads the intensity value of the self sensor. Intensity is the inverse of the distance to the closest robot module at the relative direction direction. The intensity is zero if there is no robot module in that direction, and the intensity is 1 if there is a module in the adjacent square. This function returns the intensity value.
- (ReadSensorSelfBroken direction) reads the intensity value of the SelfBroken sensor. Intensity is the inverse of the distance to the closest broken robot module at the relative direction direction. The intensity is zero if there is no broken robot module in that direction, and the intensity is 1 if there is a broken robot module in the adjacent square. This function returns the intensity value.
- (ReadSensorWall direction) reads the intensity value of the wall sensor. Intensity is the inverse of the distance to the wall or obstacle in the relative direction direction. The intensity is zero if there is no wall or obstacle in that direction, and the intensity is 1 if there is a wall block in the adjacent square. This function returns the intensity value.
- (Rotate direction) rotates the robot module by the amount direction. This does not physically move the module, it just resets the internal state of the robot module's internal facing direction. This function returns the value of direction.
- (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 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 sent.
- (SetMemory index value) sets the value of the robot module's memory numbered index to value. This function returns value.

4.4 Terminal set

The following terminal set is used by all the problems presented in this paper except the move-to-goal problem.

BASIC_TERMINAL_SET = {`GetTurn`, `0.0`, `0.25`, `0.5`, `0.75`, `1.0`, `-1.0`}

The terminal set for the move-to-goal problem has three additional terminals.

MOVE_TO_GOAL_TERMINAL_SET = BASIC _TERMINAL_SET + {`GoalDirection`, `GoalX`, `GoalY`}

An explanation for each of the terminals follows.

- `(GetTurn)` returns the current value of the "turn" variable for the simulation. The "turn" variable 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.
- `(GoalDirection)` returns the direction number corresponding to the direction from the current robot module to the goal in the module's frame of reference.
- `(GoalX)` returns the X distance from the current module to the goal in the module's frame of reference.
- `(GoalY)` returns the Y distance from the current module to the goal in the module's frame of reference.

4.5 Fitness

The fitness for each of the six problems is computed by running a simulation of the robot's actions in the world. The simulation is run for a predetermined number of turns. In each turn, the evolved program tree is executed one time for each module, as shown in this code fragment:

```
for (turn = 0; turn < maxTurns; turn++)
    for (module = 0; module < maxModules; module++)
        programTree.eval(module);
```

While executing the program tree for a single module, the sensor readings, the primitives that access the internal state of the modules, and the movement primitives are all executed for that one module. The simulation for the bridge problem was run for 160 turns. For the other four problems, the simulation was run for 120 turns. At the beginning of each fitness evaluation, the values of the communication receive buffers and the values in all the robot's memory locations are initialized to zero.

Passage problem: The fitness is the sum of penalties computed after each module is executed in each turn of the simulation. The penalty is the sum of the distances of all the robot modules to the top of the exit row of the passage. This fitness function penalizes the distances of all the modules at each point in the simulation, so it rewards the robot for solving the problem quickly.

Switchback problem: The fitness is computed in the same way as in the passage problem.

Passage and carry broken module problem: The fitness is computed in the same way as in the passage problem. One of the modules is broken so it cannot initiate a move or communicate with other modules. Solving this problem requires the robot to carry the broken module through the passage.

Passage and eject broken module problem: The fitness function has the same penalties for the working modules as passage problem. The contribution to fitness for the broken module is the distance between the broken module and the bottom of the world. This penalty is added to the fitness function after each module is simulated, so the robot is rewarded for ejecting the broken module quickly.

Bridge problem: The fitness is computed at the end of the simulation. The fitness is the sum of the distances between each module and the top of the world. The robot must configure itself into a bridge to cross the gap.

Move to goal problem: The fitness is computed at the end of the simulation, and combines two penalties. The first penalty rewards the robot for getting close to the goal by taking the average percentage distance remaining between each robot's starting location and the goal. The second penalty penalizes the robot for getting disconnected by taking the percentage of modules that are disconnected from the closest connected group of modules to the goal. The fitness is the first penalty plus twice the second penalty.

4.6 Parameters

The parameter values for all six problems are the same except for population size. Population sizes for each problem are given in the results section. The crossover and mutation rates are 99%, and 1% respectively. Crossover selects nodes uniformly from all the nodes in a tree. 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 [Koza 1992]. The maximum depth for program trees in generation 0 is 6. The maximum depth for program trees created by crossover and mutation is 10. The maximum depth of subtrees created by

mutation is 4. We use strongly typed genetic programming [Haynes 1995].

4.7 Parallelization

The population is divided into semi-isolated subpopulations called demes following Wright [Wright 1943]. Breeding is panmictic within each deme, and rare between demes. The parallelization scheme is the distributed asynchronous island approach [Bennett 1999]. 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.

4.8 Termination

For the bridge problem, the run is terminated when all the modules have made it across the bridge and up to the top of the world. The other five problems are run until they make no further progress, as the optimal fitness is not known.

5 Results

5.1 Switchback problem

This problem was run with 20 demes and 2,500 individuals per deme, for a total population of 50,000. This problem solved on its one and only run.

The best program evolved for this problem has a fitness of 12,770 and appeared in generation 156 of the run. This program has 73 instructions.

```
(If (MoveLine (Rotate (Divide (SetMemory (GetMemory 0.75) (Rotate (GetMemory GetTurn))) (SetMemory (SetMemory -1.0 GetTurn) (Rotate (GetMemory GetTurn)))))) 0.5 (If (MoveLine (Rotate (Divide (SetMemory (GetMemory GetTurn) (Rotate (GetMemory GetTurn))) (SetMemory (SetMemory -1.0 GetTurn) (Rotate (GetMemory GetTurn)))))) (GetMemory (SetMemory (Rotate (Divide 0.0 0.75)) (Rotate (GetMemory GetTurn)))) (If (MoveLine (Divide (GetMemory GetTurn) (SetMemory (Rotate (Divide 0.0 0.75)) (Rotate (GetMemory GetTurn))))) (Divide 0.5 0.0) (If (MoveLine 0.25) (SetMemory (Divide 0.5 0.0) (Rotate (Divide 0.0 0.75))) (Rotate 0.25)))))
```

The qualitative behavior of this evolved controller is as follows. The modules quickly aligned their facing directions to all face East as they used `MoveLine` to head towards the passage. They moved efficiently toward the passage opening and down the first leg of the passage. The remaining motion was still fairly efficient, with just occasional backtracking. The robot stayed in one piece until it began to go round the third bend, at which point it split into two pieces, the one in front containing 3 modules, the one in back 6. A few moves later two of the modules in back broke off from the group of six and joined up with the front group of 3 just as they exited the passage. The remaining four modules exited the passage together.

The modules faced East down the first leg of the passage, and north while moving along the first jog, in both cases facing 90 degrees away from the primary direction of motion. Similarly along the third leg, the modules faced West. For the final two legs the facing directions among the different modules were not consistent.

5.2 Passage and carry broken module problem

This problem was run with 6 demes and 2,500 individuals per deme, for a total population of 15,000. This problem solved on its one and only run.

The best program evolved for this problem has a fitness of 7786 and appeared in generation 476 of the run. This program has 65 instructions.

```
(SetMemory (ReadSensorSelfBroken (Rotate (Divide (Divide 1.0 0.75) (GetMemory (ReadSensorSelf (ReadSensorSelf 1.0)))))) (If (MoveLine (SendMessage (SendMessage GetTurn 0.25) 0.25)) (Rotate (Modulus (ReadSensorSelfBroken 0.75) 0.0)) (If (And (And (LT 0.0 0.25) (MoveLine 0.0)) (And (MoveLine (SendMessage GetTurn 0.25)) (And (MoveLine (Divide (Divide 1.0 0.75) (GetMemory 0.25))) (MoveLine 0.25)))) (If (MoveLine 0.25) (Rotate (Modulus 0.75 0.0)) (ReadSensorSelf 1.0)) (Modulus (If (MoveLine 0.5) (ReadSensorWall 0.25) (SendMessage 0.25 GetTurn)) 0.75))))
```

The modules align to face East while moving straight North. Once they hit the wall, they move East towards the passage opening, still remaining in a 3 x 3 formation. Once they reach the passage opening, one module moves to the left of the opening while the others go inside the passage. The module to the left of the opening later pushes the broken module into the passage. These two modules plus a third became disconnected from the rest of the robot before entering the passage, but join up with the others within the first leg. Along the second leg, the broken module is first pulled away from the corner by a module in front using a `MoveLine`, then three modules line up behind to help

push it along. The broken module is again pulled by a module into the final leg and then pushed by modules behind it.

5.3 Passage and eject broken module problem

This problem was run with 4 demes and 2,500 individuals per deme, for a total population of 10,000. This problem solved on its one and only run.

The best program evolved for this problem has a fitness of 17,343 and appeared in generation 29 of the run. This program has 63 instructions.

```
(If (MoveLine (Rotate (Modulus (ReadMessage (Rotate (Modulus 0.75 0.0))) (If (MoveSingle (Divide 0.75 -1.0)) (ReadSensorSelfBroken (Modulus -1.0 0.5)) (GetMemory (SetMemory GetTurn 0.25)))))) (ReadSensorWall (SetMemory 0.0 0.0)) (If (MoveLine (Modulus 1.0 (Modulus (Modulus 0.75 0.0) (If (LT (Rotate 0.25) 0.0) 0.5 (ReadMessage (SetMemory 0.0 0.0)))))) (SetMemory GetTurn 0.25) (ReadMessage (SetMemory (If (MoveLine (Modulus 1.0 0.75)) -1.0 (Modulus -1.0 (Rotate (Modulus 0.0 (ReadMessage GetTurn))))) 0.0))))
```

The robot initially moves East in a 3 x 3 formation until it is in line with the passage opening. Most of the modules head straight to the passage opening, but the three Southwest modules break off from the rest. The southmost module pushes the broken module forward twice before going around it to the left, and another module pulls it forward once, so the end state is not completely optimal. All of the modules face North once they have left the broken module behind. The robot is in three pieces as it enters the passage. Within the passage the groups reconnect, but separate again into two groups before leaving the passage.

5.4 Bridge problem

This problem was run with 4 demes and 4,000 individuals per deme, for a total population of 16,000. This problem solved on its one and only run.

The best program evolved for this problem has a fitness of 9 and appeared in generation 60 of the run. This program has 49 instructions.

```
(If (And (And (MoveLine 0.25) (MoveLine (If (And (MoveLine 0.25) (MoveSingle 0.5)) (GetMemory (SetMemory 0.25 1.0)) (GetMemory (SetMemory 0.25 1.0))))) (And (MoveLine 0.25) (MoveLine 0.5))) (SetMemory (If (And (MoveLine 0.25) (MoveSingle 0.5)) (SetMemory (SetMemory (Modulus GetTurn 0.25) 0.0) (Rotate -1.0)) (GetMemory (SetMemory 0.75 1.0))) (ReadSensorWall 0.25)) (Modulus GetTurn 0.25))
```

The robot remains in a 3x3 formation as it heads straight North to the wall and then West to the gap opening. The front row enters the gap one by one, as other modules move forward to the wall. The modules cross the gap in single file. Once on the other side, the frontmost module moves left so that it can pull the others across. The robot stays in on piece until it has crossed the gap (that is the only way it can solve the problem). The robot then breaks into two pieces before forming a single line against the North wall. The modules face East throughout.

5.5 Move to goal problem

This problem was run with 1 deme of 2,000 individuals. The problem was run 20 times, and we report here on the best solution found of those 20 runs.

The best program evolved for this problem has a fitness of 0.1248 and appeared in generation 20 of the run. This program has 16 instructions.

```
(If (MoveLine GoalDirection) GoalLocationX (If (MoveLine (If (MoveSingle (Rotate 0.25)) GoalDirection GoalLocationX)) (Rotate 0.25) (ReadSensorSelf 0.25)))
```

This program first tries to move the current module in the goal direction. If that succeeds the program is done. Otherwise, the program rotates the module 90 degrees to the left and attempts a `MoveOne`. If the `MoveOne` succeeds, the module then also attempts a `MoveLine` in the goal direction. If the `MoveOne` fails, the module attempts a `MoveLine` in the direction of the X displacement to the goal. If the `MoveLine` succeeds, the module then rotates 90 degrees to the left.

6 Comparisons

We performed 6 sets of comparative runs for the passage problem to determine the effects of changing various run parameters. All runs used a population of 3000. The base case for all comparisons consisted of 40 runs of the passage problem as described above.

The first study involved 37 runs of the passage problem with the `MoveOne` primitive removed from the function set. The second study involved 40 runs of the passage problem with the `MoveLine` primitive removed. As can be seen in Figure 7, by generation 100, 45% of the base case runs had solved, 73% of the runs without the `MoveOne` function had solved, and only 5% of the runs

without the `MoveLine` function had solved. The `MoveOne` function represents the raw basic moving capability of the hardware, whereas the `MoveLine` function is a higher level movement abstraction. The results illustrate the advantage of including higher level movement abstractions.

The third study involved 40 runs of the passage problem using 100% mutation and no crossover, while the fourth study used 90% crossover and 10% cloning in 47 runs. As can be seen in Figure 8, by generation 100, 67.5% of the runs without crossover had solved, 55.3% of the runs with 10% cloning had solved, compared to only 45% of the base cases. We do not know why crossover was not helpful for this particular problem.

The fifth study involved 48 runs of the passage problem with the `Prog2` function added to the function set. The sixth study involved 37 runs of the passage problem with the following six functions added to the function set: `Add`, `Sub`, `Mult`, `GreaterThanOrEqual`, `Or`, and `Not`. As can be seen in Figure 9, the addition of the `Prog2` function was some advantage in the middle generations, but by generation 100 there was little difference compared to the base case. The runs with the additional six functions kept pace with the base case up through generation 60, but then fell behind.

7 Conclusions

The enormous increase in difficulty for the passage problem when the `MoveLine` function was removed suggests that higher level movement functions beyond the basic single module move directly supported by the hardware are important for this type of problem. We plan to investigate whether genetic programming can discover such higher level movement functions automatically using Automatically Defined Functions [Koza 1994].

None of the solutions exploited the communications capabilities provided; none used both `SendMessage` and `GetMessage`. Either the tasks we attempted were too simple for communication to be helpful, or the capabilities we provided need to be redesigned in order to be useful.

We have demonstrated the automatic design of distributed controllers that are capable of correctly completing a task even when one of the robot modules is broken. We have demonstrated two different ways that broken modules can be handled: carrying the broken module as part of the configuration, and reconfiguring to remove the broken module. We have demonstrated a uniform approach for automatic distributed controller design that can be used for several different kinds of problems.

8 Future work

We will apply our basic approach to a variety of modular robots, including three dimensional modules and modules that use non-sliding moves such as compression and expansion moves [Rus 1999], and rotational moves [Bojinov 2000].

We have used small numbers of modules in these experiments. Larger systems are of greater interest. The simulation times for the fitness evaluation for much larger systems can be too long to be practical. We will investigate

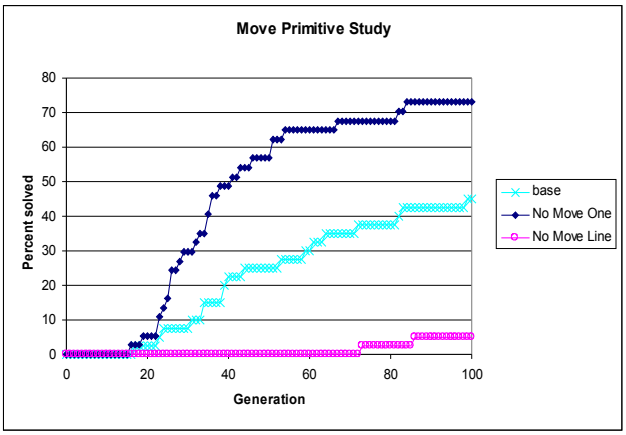

Figure 7 The percentage of runs solved by generation for the base case runs, the runs without `MoveOne` in the function set, and the runs without `MoveLine`.

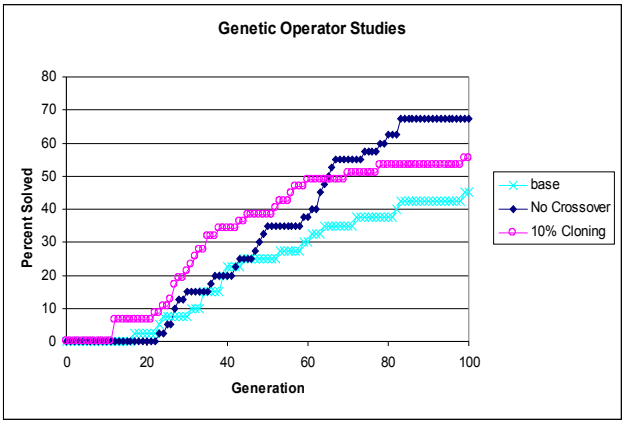

Figure 8 The percentage of runs solved by generation for the base case runs, the runs with crossover disabled, and the runs with cloning enabled.

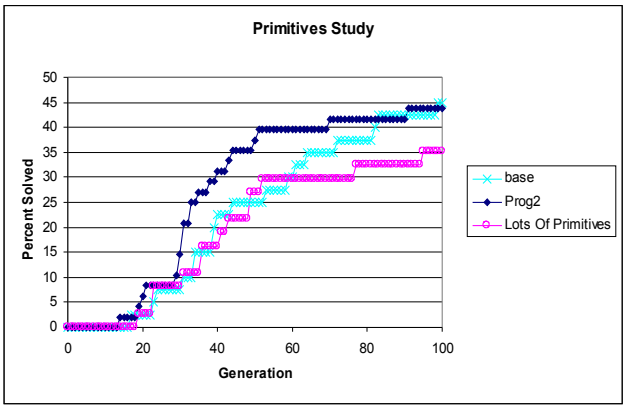

Figure 9 The percentage of runs solved by generation for the base case runs, the runs with Prog2 added, and the runs with six additional functions.

techniques for evolving controllers that extend correctly to robots with larger numbers of modules.

Acknowledgments

Many thanks to Tad Hogg and Arancha Casal for teaching us about modular robots and suggesting problem areas for us to work on. Thanks also to Wolfgang Polak for help in setting up the computing cluster and for conversations about modular robots. We are grateful to Adil Qureshi for his Gpsys 1.1 java genetic programming source code on which our system is based.

References

[Bennett 1999] Bennett, Forrest H III, Koza, John R., Shipman, James, and Stiffelman, Oscar, "Building a parallel computer system for $18,000 that performs a half peta-flop per day" in Banzhaf, Wolfgang, Daida, Jason, Eiben, A. E., Garzon, Max H., Honavar, Vasant, Jakiela, Mark, and Smith, Robert E. (editors). GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999, Orlando, Florida USA. San Francisco, CA: Morgan Kaufmann.

[Bonabeau 1999] Bonabeau, Eric, Dorigo, Marco, Theraulaz, Guy, *Swarm Intelligence: from Natural to Artificial Systems*, Oxford Univ. Press, 1999.

[Bojinov 2000] Hristo Bojinov, Arancha Casal, Tad Hogg, "Emergent Structures in Modular Self-reconfigurable Robots", IEEE Intl. Conf. on Robotics and Automation (ICRA) 2000.

[Casal 1999] Arancha Casal, Mark Yim, "Self-Reconfiguration Planning for a Class of Modular Robots", Proceedings of SPIE'99, Volume 3839.

[Chen 1996] I. M. Chen, "On Optimal Configuration of Modular Reconfigurable Robots," 4th International Conference on Control, Automation, Robotics and Vision, Singapore 1996.

[Haynes 1995] Haynes, Thomas, Wainwright, Roger, Sen, Sandip, Schoenfeld, Dale, "Strongly Typed genetic Programming in Evolving Cooperation Strategies", in Proceedings of the Sixth International Conference on Genetic Algorithms, San Francisco, CA: Morgan Kaufmann. 271 - 278.

[Koza 1992] Koza, John R, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: MIT Press.

[Koza 1994] Koza, John R., *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: MIT Press.

[Kotay 1998] Kotay, K., Rus, D., Vona, M. and McGray, C., "The Self-reconfiguring robotic molecule: design and control algorithms". In Proceeding of the 1998 International Conference on Intelligent Robots and Systems, 1998.

[Pamecha 1996] Pamecha, A., Chiang, C. J., Stein, D., Chirikjian, G. S., "Design and Implementation of Metamorphic Robots." In Proceedings 1996 ASME Design Engineering Technical Conference and Computers and Engineering Conference, 1-10. New York, NY, ASME Press, 1996.

[Pamecha 1997] Amit Pamecha, Imme Ebert-Uphoff, Gregory S. Chirikjian, "Useful Metrics for Modular Robot Motion Planning," IEEE Transactions on Robots and Automation, pp. 531-545, Vol.13, No.4, August 1997.

[Rus 1999] Rus, D., M. Vona, "Self-Reconfiguration Planning with Compressible Unit Modules". In 1999 IEEE Int. Conference on Robotics and Automation.

[Tummala 1999] Tummala, R. L., Mukherjee, R., Aslam, D., Xi, N., Mahadevan, S., and Weng, J., "Reconfigurable Adaptable Micro-Robot", Proc. 1999 IEEE International Conference on Systems, Man, and Cybernetics, Tokyo, Japan, October 1999.

[Ünsal 2000a] Ünsal, C., H. Kiliççöte, and P. K. Khosla, "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.

[Ünsal 2000b] Ünsal, C., H. Kiliççöte, Mark Patton, and P. K. Khosla, "Motion Panning for a Modular Self-Reconfiguring Robotic System," submitted to 5th International Symposium on Distributed Autonomous Robotic Systems (DARS 2000), October 4-6, 2000, Knoxville, Tennessee, USA.

[Wright 1943] Wright, Sewell. 1943. Isolation by distance. Genetics 28:114-138.

[Yamasaki 1998] Yamasaki, Koetsu, Kundu, Sourav, Hamono, Michitomo, “ Genetic Programming Based Learning of Control Rules for Variable Geometry Structures,” In the Proceedings of the 3rd Annual Genetic Programming Conference, 1998, Morgan Kaufmann Publ., San Francisco, 1998.

[Yim 1997] Yim, Mark, Lamping, John, Mao, Eric, Chase, J. Geoffrey, "Rhombic Dodecahedron Shape for Self-Assembling Robots", Xerox PARC SPL TechReport P9710777, 1997.

[Yim 2000] Yim, Mark, Duff, David G., Roufas, Kimon D. IEEE Intl. Conf. On Robotics and Automation (ICRA) 2000.