Collaborating with a Genetic Programming System to Generate Modular Robotic Code

Jeremy Kubica*
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15232
jkubica@ri.cmu.edu

Eleanor Rieffel
FXPAL
3400 Hillview Ave Bldg 4
Palo Alto CA 94304
rieffel@fxpal.com
(650) 813-7077

Abstract

The choice of a set of primitives can strongly effect the performance of genetic programming. The design of such a set remains more of an art than a science. We look at a joint approach that combines both hand-coding and genetic programming to define and refine primitives in the context of a creating distribute control code for modular robots. We give some rules of thumb for designing and refining sets of primitives, illustrating the rules with lessons we learned in the course of solving several problems in control code for modular robots.

1 INTRODUCTION

It is well-known, from various "no free lunch" theorems, that no single approach can effectively solve all problems (Wolpert, 1996; Wolpert and Mecready, 1997). Thus it is necessary to tailor techniques to the problem at hand. For genetic programming this tailoring can be done in three primary ways: choosing the primitives, choosing the fitness functions, and choosing parameters for the algorithm such as the percentage of use of various operators. Here we look at the problem of effectively choosing primitives.

We describe techniques for and experiences with the joint use of genetic programming and hand-coding to design and refine sets of primitives for a variety of modular robotics problems. The creation of decentralized control software for modular robots is a difficult problem due to both the decentralized nature of the software and the fact that the connectivity relations between the modules constantly change. Yet, in spite of these difficulties, or perhaps because of them, modular robotics provides an ideal domain in which to experiment with automated software generation methods; there are many robotic tasks that are easily specified but for which it is highly non-obvious what distributed software would create the desired behavior.

Ideally, for the problem of robotic control, a set of primitives could be derived directly from a description of the hardware capabilities. While a description of such capabilities is an excellent place to start, finding a solution given only the most basic actions can be prohibitively time consuming. Another approach is to tailor the primitives to the task, which may require significant development time. In fact, the development time required for the primitives can approach that of actually solving the problem. We take an intermediate approach. Except for primitives that encode information specific to the problem (as opposed to its solution), we try to provide primitives that would appear to be useful in a wide range of problems and are still reasonably low-level. We use insights gained from both human and machine attempts to solve the problems to design effective sets of primitives.

2 MODULAR ROBOTIC HARDWARE AND SIMULATOR

2.1 TELECUBE MODULES

Modular self-reconfigurable robots are systems consisting of a collection of simple and identical robotic modules that can form connections to and move relative to each other (Yim, 1994; Murata et. al., 1994; Rus and Vona, 2000). These modules function together to produce an overall behavior of the robot, analogous to the way cells in a body function together. The fact that both the connectivity and relative positions of the modules can change allows the overall robot to reconfigure and take on different shapes. Modular robotic

* Supported by FXPAL

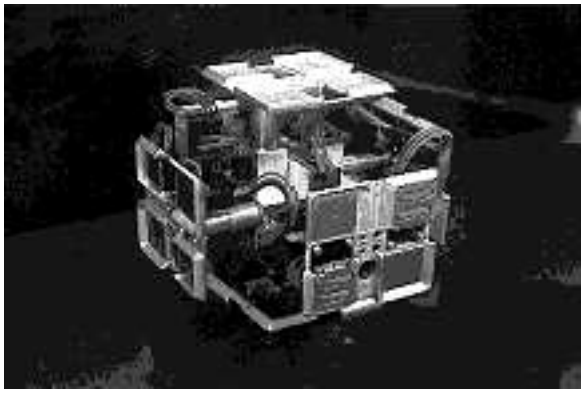

Figure 1: One telecube module.

systems provide possible benefits in terms of flexibility, adaptability, and robustness.

2.2 MODULE CAPABILITIES

2.2.1 Physical Capabilities

The robot modules we consider are the TeleCube modules currently being developed at Xerox PARC and shown in Figure 1 (Suh et. al., 2002). They are similar in design to modules being designed and build at Dartmouth (Rus and Vona, 2000), which can expand in two dimensions. The modules are cube shaped with an extendable arm on all six faces. The arms are assumed to extend independently up to half of the body length, giving the robot modules an overall 2:1 expansion ratio. For simplicity, we restrict the arms to fully extended or fully retracted states. The expansion and contraction of these arms provide the modules with their only form of motion. Latches on the plates at the end of each arm allow two aligned modules to connect to each other. The arm motion together with the latching and unlatching capability means that the connectivity topology of a modular robot can change greatly over time. As shown in (Vassilvitskii et. al., 2002), this motion is sufficient to enable arbitrary reconfiguration within a large class of shapes.

2.2.2 Sensors, Communication, and Memory

Each module is assumed to have simple sensing and communication abilities that resemble those that will be given to the TeleCube modules currently being built at Xerox PARC. Modules can send limited bandwidth messages to their immediate neighbors. Each module is also assumed to be able to sense contact at the end of each arm, sense how far each of their arms is extended, determine whether they are connected to a neighboring module, and detect nearby, adjacent modules. Each module has a small memory capacity, which is initialized to all zeros. We also give the modules simple computational abilities such as the capability to determine the opposite of a given direction, the ability to generate a random direction, and to calculate and store the value of each of its position coordinates.

2.3 TELECUBE SIMULATOR

The experiments reported here were run by connecting FXPAL's genetic programming system to a simulator, written by J. Kubica and S. Vassilvitskii, for the TeleCube modular robots. The module control code is represented in a LISP-like parse tree that is evaluated once per time step for each module.

3 PROBLEM CONTEXTS

For all of the problems described here we are interested in a completely decentralized solution in which the desired global behavior of the robot emerges from control code that is run locally on each of the modules that make up the robot. Decentralized control software is a challenging domain to begin with, and the fact that the connectivity relations between the modules constantly change makes modular robotic problems even harder. The collaboration between hand-coding and automatic generation of solutions is described below in the context of three specific modular robotic control problems to which we have applied this technique: the tunnel problem, the filtering membrane problem, and the sorting membrane problem.

The *tunnel problem* consists of a long thin world, 40 x 10 x 2 arm lengths, that is enclosed on all sides by walls. During each of the trials an object is placed randomly "in" one of the long walls. This means that a module adjacent to this location along the wall can sense the object, but will not be blocked by it or stuck behind it while moving along the wall. The goal of the modules is to find the object and all move as close to it as possible. Thus, we define our fitness function as the sum of each module's distance to the object at the end of a run. The modules start out as a 3 x 3 x 1 grid configuration in one corner of the world.

The membrane problems consist of a membrane, a three-dimensional lattice of modules, and a foreign object which, depending on its attributes, should or should not be accepted into the membrane and manipulated by it. In the case of a *filtering membrane*, the object is either accepted or rejected. Accepted objects are passed through the membrane and out the bottom, whereas rejected objects remain on top. A *sorting membrane* accepts all objects, but sorts them

along some axis. We looked at the case of a binary sorting membrane, which moves an object towards one of two opposite ends depending on its value. For both membrane problems we use fitness functions that measure the distance of the object from where it ought to end up and a penalty for the membrane breaking apart. Since the modules are unable to directly grasp or pull the objects, all solutions to the membrane problem require the use of gravity, by having modules move out from under the objects and allowing them to fall through the membrane. For more details on the membrane problems, including some pictures, see (Kubica and Rieffel, 2002).

4 RELATED WORK

A number of researchers have looked at enabling a genetic programming system to add primitives on its own in the course of its runs. Approaches include ADFs (Koza, 1994), libraries (Angeline and Pollack, 1994), subroutines (Rosca and Ballard, 1996), and subtree encapsulation (Roberts et al., 2001). While such techniques might help with some of the problems we describe here, we are particularly interested in how to add primitives that the system would be unlikely to find for itself. Furthermore, there are many situations, perhaps the majority, in which one is most interested in solving the problem at hand in any way possible, rather than being concerned about the extent to which the solution was automatically attained. We hope that what we write here can help others to have more productive collaborations with a genetic programming system in order to more effectively solve practical problems of interest.

The problem of automatic code generation for modular robots has also seen interest recently. Kubica et. al. (Kubica et. al., 2001) hand-coded control programs for internal object manipulation with robots made of TeleCube modules. Bennett et. al. (Bennett et. al., 2001) used genetic programming to generate distributed control programs for modular robots consisting of sliding-style modules (Bennett and Rieffel, 2000; Pamecha et al., 1996). It is important to note however that this sliding-style module design enables movement with primitive operations directly suggested by the hardware. In particular, movement in that setting, unlike for the TeleCube modules, does not require explicit connection and disconnection actions. Thus movement in their case avoids some of the difficulties we faced when attempting to generate effective software for robots made from Telecube style modules.

5 DEVELOPING EFFECTIVE SETS OF PRIMITIVES

5.1 BASIC PRIMITIVES

Each of the basic capabilities of the modules can be captured by a primitive operation.

- Physical actions: `(ExtendArm direction)`, `(RetractArm direction)`, `(Connect direction)`, `(Disconnect direction)`
- Communication: `(SendMessage direction type value)`, `(GetMessage direction type)`
- Sensors: `(HasNeighbor direction)`, `(ReadSensorNeighborDist direction)`, `(ReadSensorObjectDist direction)`
- Memory: `(ReadReg index)`, `(SetReg index value)`
- Other: `(OppDirdirection direction)`, `(RandDir)`, and `(GetX)`, `(GetY)`, `(GetZ)`.

In addition to these module specific primitive, we allow the module control programs to use the following basic programming primitives: `(Add)`, `(Sub)`, `(If)`, `(ProgN)` `(LT)`, `(And)`, and `(Not)`, and the numeric constants `0.0`, `0.1`, `0.2`, `0.4`, `0.6`, `0.8`, and `1.0`.

The primitives described above all mirror basic hardware capabilities of Telecube modules. For more details on these primitives, see (Kubica and Rieffel, 2002). However these primitives do not form an optimal set of primitives for either a human or a genetic programming system to construct effective software for the tasks we described above. We look at how genetic programming experiments and hand-coding attempts together enabled the development of effective sets of primitives.

5.2 INITIAL CHOICE OF PRIMITIVES

Ask the system only to discover a solution, not aspects of the problem

One important consideration when choosing primitives is to give the system full information about the problem. Even researchers more concerned with automatically generating code than solving problems per se and who are resistent to giving the system any hints as to how to solve the problem, should feel comfortable giving the system enough information so that it does not have to guess the problem as well as the solution.

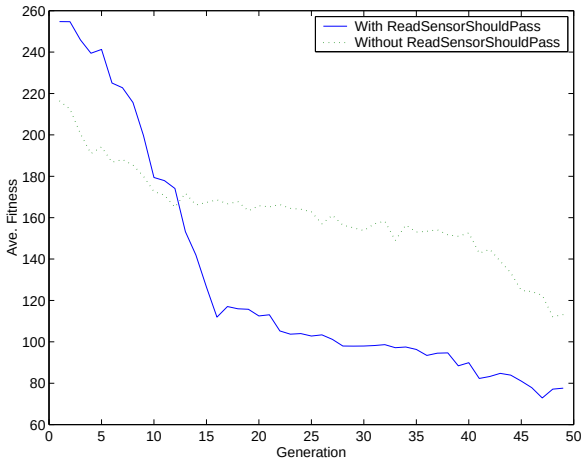

Figure 2: Average solution performance on filtering membrane problem with and without ReadSensorShouldPass primitive.

In the filtering membrane problem, the modules must determine whether or not to accept an object. In our case the criterion for the filter to accept an object was that the value returned by `(ReadObjectVal direction)` be above `0.5`. In order to solve the problem in our initial runs the GP system had to create a subtree testing for the acceptance criterion.

As we tried to gauge how the system was doing, we realized that we were using the existence of such a subtree to determine how close the GP system was to a solution. We soon realized that one should not require an automatic code generation technique to guess the problem as well as the solution, even if it can. Thus we added a primitive that encapsulates the filtering criterion: the Boolean `(ReadSensorShouldPass direction)` primitive

`(LT(Add(0.4 0.1))(ReadObjectVal direction))`.

Figure 2 shows the performance of two GP runs, one with the `ReadSensorShouldPass` primitive and one without it. While the system could solve the problem without the `(ReadSensorShouldPass direction)` primitive, its addition certainly helped. Further, for the evolution of the membrane control there is no reason to withhold this type of information about the problem statement itself. Note that this situation is different from one in which one might be trying to learn a good criterion, say for identifying defective parts or a distinguishing characteristic of a set of objects.

Avoid large needle-in-the-haystack searches

The tasks we describe above all require movement, as

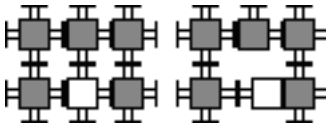

Figure 3: Module in white moves towards right.

do most modular robotic tasks including reconfiguration, locomotion, and internal manipulation. But none of the fitness functions we used give any reward to programs that have put together part, but not all, of a sequence that would result in movement. Since no reward is given, the genetic programming system has nothing to guide it towards movement.

Even at its simplest level, movement of a single TeleCube module is a complex process, involving bonding with and unbonding from neighbors, expanding and contract arms, and checking that the move is possible. To get an impression the complexity of a single movement, it is only necessary at look at the movement subtree involved in disconnecting from the perpendicular neighbors:

```
(ProgN (ProgN (Disconnect 0.0) (RetractArm
0.0)) (ProgN (Disconnect 0.2) (RetractArm
0.2)) (ProgN (Disconnect 0.4) (RetractArm
0.4)) (ProgN (Disconnect 0.6) (RetractArm
0.6)))
```

The code above disconnects from the correct neighbors in the case the module is moving along the $\pm$X axis. Just to choose `Disconnect` and `Retract` arm from just the four physical primitives has probablility less than 10^{-3}. Choosing the appropriate constants for the `Disconnect` and `Retract` functions from the seven numeric constants also has probability less than 10^{-5}. And choosing the five `ProbN` primitives has probability less than 10^{-5}. So the probability of finding the correct components is less than 10^{-13}.

Up to now we have only considered a subtree of the sort that would correctly disconnect a module. In addition, one would need appropriate `if` statements to determine the correct direction, the appropriate arm expansions and contractions to accomplish the movement, and a similar subtree to handle reconnection with any new neighbors. Thus, the simple move illustrated in Figure 3, given only the basic primitives described above, would require at least 12 different primitives and well over 50 nodes. More code would be required to enable moves in all six directions, not just one, and the graceful handling of movement failures would require considerably more code. Thus the likelihood that such a program would be created given

no guidance is impractically small.

With the basic primitives, and a simple fitness function with no gradient towards movement from none, the GP system would be searching blindly. Thus it is extrodinarily unlikely that the GP system would succeed even in evolving movement. In fact the problem is harder still since only useful and used motion would be rewarded. By asking genetic programming to evolve movement, we are asking it to search for a needle, or rather a few needles, in a very large haystack.

A number of solutions are possible. One solution would be to change the fitness function so that it rewards partial programs out of which movement could be constructed. Another possibility is to simply evolve movement first and then use that solution as a primitive when evolving solutions to more complex tasks. Note that this approach still requires determining a fitness function that rewards partial solutions. It is highly non-trivial to see how to design a principled reward system. Figuring out principled approaches to this problem, and other similar problems, is an open research issue that ought to be of great interest to anyone trying to automatically solve problems. An unprincipled way of solving this problem is to hand-code a solution and then reward programs based on similarity with this solution. One might as well use the hand-coded solution as a primitive in the first place, which is what we do.

Achieving simple motion, for instance the motion illustrated in Figure 3, only requires reasoning at the local level and is thus easier for a human to do than solutions to tasks that require global behavior to emerge from the local actions. Following basic operations used by Kubica, et. al. (Kubica et. al., 2001), we created a movement primitive that enables modules to move a distance of one arm length in any of the six directions. The movement is accomplished by simultaneously contracting the front arm and expanding the back arm to effectively "slide along the arms." The movement primitive also checks whether movement is possible, reverses any steps taken prior to a failed check, and returns whether or not the movement has succeeded.

This example also serves to illustrate another principle useful for effective collaboration with a GP system in solving a problem.

Hand-design parts that are easy for a human to write but hard for a GP system to discover.

5.3 REFINING THE PRIMITIVES

Delete or replace unused primitives

If the system is solving a problem without certain primitives consider removing or replacing them. Use this information to update your intuition if possible. Encapsulating such primitives is a particularly attractive choice in a number of situations.

Encapsulate hard-to-use primitives

If the system is not using a primitive, or a set of primitives, effectively, consider giving the system higher level primitives that it may be able to use more easily. The best time to encapsulate actions into primitives is when a certain subtree is needed.

The clearest case for removing a primitive is when a higher level primitive replaces it. For example, the implied TeleCube primitive `(Disconnect direction)` is automatically handled by such primitives as `(Move direction)` and `(RetractArm direction)`. Further, maintaining global connectivity is vital for power routing and facilitates alignment of connecting modules and inter-module communication. Thus, we wish to maintain a high level of connectivity at all times and additional uses of disconnection may not only be unnecessary, but also detrimental. Thus, it can be said that the disconnect primitive was replaced by incorporating it into higher-level primitives.

A second example of the removal of primitives is in the more complex problem of module communication. Since control is local, it would seem that communication between the modules would be required in order for the robot to achieve a task of any complexity. Messages would enable modules to share information and coordinate actions. However, communication may not be as necessary for modular robotic tasks as humans tend to think. Bennett and Rieffel comment in their conclusions that none of the solutions to any of the five tasks they studied (Bennett and Rieffel, 2000) used the communication capabilities provided. Similarly, while the solution given in (Bennett et. al., 2001) contains two `(SendMessage)` commands, since it contains no `(ReadMessage)` commands, one can deduce that the communication capabilities were not used.

The lack of use of the communication primitives in solutions to various problems suggest that a wider class of problems can be solved without communication than most humans would generally think. However, it also suggests that the provided communication primitives may be hard for a genetic programming system to use. Effective communication requires the generation of a quantity that would be useful to communicate, the sending of that quantity, the receipt of that quantity, and the use of that quantity by the receiver. In most cases, until all of these steps are in

place and correct there is no benefit. So for a genetic programming system, with a simple fitness function in which there are no explicit rewards for partial communication, the generation of effective communication is reduced to a large needle-in-the-haystack problem similar to the movement problem we described above. Unfortunately, in this case it is less clear what to do by hand since what sort of communication would be useful is less clear.

One problem with the message primitives described above is that it is difficult to send out information to all neighbors or send direction dependent information, such as "move away." One interesting extension to the communications capabilities described above is the addition of gradients. Gradients are messages that are broadcast to all surrounding neighbors. These neighbors in turn rebroadcast them to their neighbors, with the strength of the gradient decayed with each broadcast.

Previous work with gradient messages in hand-coded solutions was shown effective in (Bojinov et. al., 2000; Kubica et. al., 2001) and in (Shen et. al., 2000; Shen et. al. 2000) where the gradients were called scents and hormones respectively. Of particularly interest is the use of gradients for internal manipulation of objects. The use of scents applied to the problem of internal object manipulation in (Kubica et. al., 2001) limited the scents to positive and negative, where modules were inclined to move towards the positive scent and away from the negative scent. This implies further specializing the messaging primitives to:

- `(SendPositiveGradient)` Emits a positive gradient
- `(SendNegativeGradient)` Emits a negative gradient
- `(HandleGradient)` Tries to follow the gradients, move towards positive gradient and away from negative gradient. Returns `true` if a gradient was detected and the module was able to move to follow it.

These communication primitives illustrate how human experience and intuition can be encoded in general primitives. These primitives have the potential to greatly reduce the space of programs that might be searched in order to find a solution. For example, the positive and negative gradients above do not require the use of constants, simplifying the coordination and handling of messages.

The tunnel world problem was designed as a test of the communication primitives. It is relatively easy for the modules to move randomly around the world in a group and for a single module to stop moving if it detects an object. The problem gets increasingly nontrivial when all of the modules are required to move as close as possible to the object. This implies the need for at least implicit communication between the modules. Specifically, when a module detects the object it must be able to say "move towards me" or something similar.

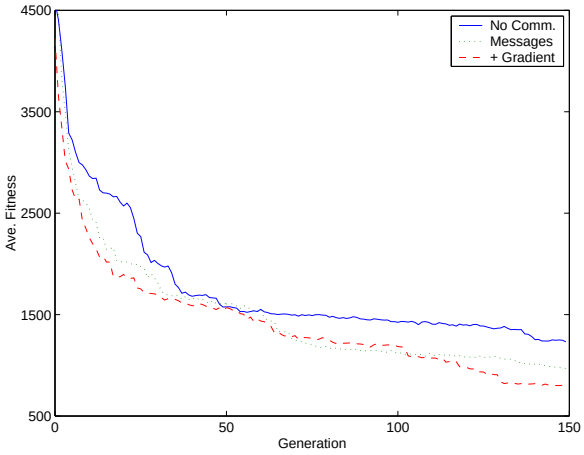

Figure 4: Average solution performance on the tunnel problem for runs with different sets of communication primitives.

To determine the effectiveness of various communication primitives, the problem was run 24 times in each of three conditions: 1) no communication primitives, 2) only the messaging primitives, and 3) only the positive gradient primitives (`SendPositive` and `HandleGradients`). The average cost for these trials during each generation of the GP runs was recorded and is shown in Figure 4. Once again, the intuitive difference is reflected in the success of the runs that were able to use the positive gradients. The solutions that can contain positive gradients perform the best after the 150 generations, while the solutions that could not have any message passing commands in them perform noticably worse.

Consider deleting unused primitives

A case can be made for keeping primitives that immediately appear to have no benefit. The argument goes that if they are not needed, GP will evolve solutions that do not use them. Despite this argument, it is important to place some restrictions on the number of primitives. As the number of primitives increases, the search space of possible programs increases. Thus one should consider deleting unused primitives, as well as

replacing or encapsulating them.

In addition to using hand-coding and genetic programming to develop primitives, the combination of techniques lead to cases where primitives were found to be ineffective and thus could be removed. For example, in the filtering membrane problem described above, the primitives `(HasNeighbor direction)` and `(ReadSensorNeighborDist direction)` were not used in early successful solutions. In addition, early attempts to hand-code the solution did not reveal any apparent benefit to them. Thus, they were removed for later runs. A viable solution was found without these primitives.

Consider adding primitives

It may become apparent after some runs that certain primitives might be necessary or helpful. In particular, revisit the points of previous section. In our case, it was at this point that we realized a movement primitive was necessary, even though that could have been clear at the start. Also, some of the gradient primitives could be viewed in this way, since it is arguable that they replace rather than encapsulate the initial message primitives.

5.4 PRIMITIVES FOR RELATED PROBLEMS

Consider using results, or partial results, from similar problems as primitives.

Solutions, or piece of solutions, found by genetic programming to similar problems, including easier versions of problems or related subproblems, can provide useful and robust primitives for more complex problems.

A sorting membrane needs to be able to pass objects through its structure just as a filtering membrane does. The solution found for the filtering membrane problem contains a quick and efficient way to pass objects through a structure of modules. By examining the solution provided by genetic programming, we can remove unnecessary sub-trees to obtain:

```
(If (ReadSensorObjectShouldPass 0.0) (If (If
(ProgN (Move 1.0) (RetractArm 0.8)) (ProgN
(ProgN (Move 1.0) (RetractArm 0.8)) (Move
1.0)) (RetractArm 0.1)) Move 1.0) (Move
0.4)) NormalizeDensity)
```

Although this program does not provide for 100% success on all problems (Kubica and Rieffel, 2002), the result is a relatively efficient filtering program. At the highest level of the primitive is an `If` clause dependent on the `ReadSensorShouldPass` primitive. By simply replacing `(ReadSensorObjectShouldPass 0.0)` with `(ReadSensorIsObject 0.0)` we were able to provide later membrane problems with a robust `(Drop)` primitive.

6 FUTURE WORK

Ideally, a system for automatically generating modular robotic code would only need a description of the capabilities of the modules and the desired behavior for the system to find a solution. Unfortunately, we are currently far from such a system. In particular, current evolutionary approachs are not sufficiently advanced to be able to solve many complex problems on their own.

One difficulty facing such systems is that they are often not capable of deriving an effective set of primitives from the information they are given. Determining fitness functions is also a nontrivial problem, particularly how to reward useful pieces that are not measured by a fitness function coming directly from a problem statement. The problem of modular robot control, particular that of generating effective communication for modular robotic systems, provides a good area in which to explore approaches to this problem.

A first step towards a more automated system is to identify situation in which current systems need help. A better understanding of how a human and a system can collaborate to effectively solve a problem, can lead not only to guidelines that can help humans solve problems more efficiently with the help of machines, but could lead to insights that could ultimately enable the automation of some of the processes currently requiring human input. Our hope is that our effort here not only contributes to this direction, but will also encourage others to do more work along these lines. These problems are difficult, but not so hard that progress cannot be made.

References

P.J. Angeline, J.B. Pollack (1994). *Coevolving High-level Representations.* Artificial Life III, Addison-Wesley, pp. 55-71.

F.H. Bennett III, B. Dolin, E.G. Rieffel (2001). *Programmable Smart Membranes: Using Genetic Programming to Evolve Scalable Distributed Controllers for a Novel Self-Reconfigurable Modular Robotic Application.* Genetic Programming: proceedings of EuroGP 2001, Springer LNCS 2038, pp. 234-245.

F.H. Bennett III, E.G. Rieffel (2000). *Design of Decentralized Controllers for Self-Reconfigurable Modu-*

lar Robots Using Genetic Programming. Proceedings of the 2nd NASA/DoD Workshop on Evolvable Hardware, pp. 43-52.

H. Bojinov, A. Casal, T. Hogg (2000). *Emergent Structures in Modular Self-Reconfigurable Robots.* 2000 IEEE International Conference On Robotics and Automation, pp. 1734-1741.

J.R. Koza (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press

J. Kubica, A. Casal, T. Hogg (2001). *Agent-based Control for Object Manipulation with Modular Self-reconfigurable Robots.* The 2001 International Joint Conference of Artificial Intelligence, pp. 1344-1352

J. Kubica, E.G. Rieffel (2002). *Creating a Smarter Membrane: Automatic Code Generation for Modular Self-Reconfigurable Robots.* To Appear in 2002 IEEE International Conference On Robotics and Automation.

S. Murata, H. Kurokawa, S. Kokaji (1994). *Self-Assembling Machine.* Proceedings of the 1994 IEEE International Conference On Robotics and Automation, pp. 441-448

A. Pamecha, D. Stein, C.-J. Chiang, G.S. Chirikjian (1996). *Design and Implementation of Metamorphic Robots.* Proceedings 1996 ASME Design Engineering Tehcnical Conference and Computers and Engineering Conference, pp. 1 - 10. ASME Press.

S.D. Roberts, D. Howard, J.R. Koza (2001). *Evolving Modules in Genetic Programming Subtree Encapsulation.* Genetic Programming: Proceedings of EuroGP2001, Springer LNCS 2038, pp. 160 - 175.

J.P. Rosca, D.H. Ballard (1996). *Discovery of subroutines in genetic programming.* Advances in Genetic Programming 2. MIT Press.

D. Rus, M. Vona (1999). *Self-Reconfiguration Planning with Compressible Unit Modules.* Proceedings of the 1999 IEEE International Conference On Robotics and Automation, pp. 2513-2520.

D. Rus, M. Vona (2000). *A Physical Implementation of the Self-Reconfiguring Crystaline Robot.* Proceedings of the 2000 IEEE International Conference On Robotics and Automation, vol. 2, pp. 1726 -1733.

W.M. Shen, B. Salemi, P. Will (2000). *Hormones for Self-Reconfigurable Robots.* Proceedings of 6th International Conference on Intelligent Autonomous Systems, pp. 918-925. IOS Press.

W.M. Shen, B. Salemi, Y. Lu, P. Will (2000). *Hormone-Based Control for Self-Reconfigurable Robots.* 2000 International Conference on Autonomous Agents. Barcelona, Spain.

J.W. Suh, S.B. Homans, M. Yim (2002). *Telecubes: Mechanical Design of a Module for Self-Reconfigurable Robotics.* To appear in the 2002 IEEE International Conference On Robotics and Automation.

S. Vassilvitskii, J. Kubica, E. Rieffel, J. Suh, M. Yim (2002). *On the General Reconfiguration Problem for Expanding Cube Style Modular Robots.* To appear in 2002 IEEE International Conference On Robotics and Automation.

D. Wolpert, W. Macready (1995). *No free lunch theorems for search.* Technical Report SFI-TR-05-010.

D. Wolpert, W. Macready (1997). *No free lunch theorems for optimization.* IEEE Transactions on Evolutionary Computation, 1(1) pp. 67 - 82.

M. Yim (1994). *Locomotion with a Unit Modular-Reconfigurable Robot.* Stanford PhD Thesis, 1994.