Astro Teller

The genetic programming process searches over a fitness landscape. The shape of this landscape is determined by the task to be solved and the *representation* in which the population members are expressed. The movement through this space is determined by the *operators* that act to recombine the population members. These factors make it imperative that our search for increased power and understanding in genetic programming include the study and improvement of representations and operators. This chapter describes a process for learning *SMART* recombination programs in a co-evolutionary process and a new representation for the evolution of algorithms. How these SMART operator programs are created, how they act, how they co-evolve with a main population of programs, and experimental results on their use are the subjects of this chapter.

3.1 Introduction

Traditional formulations of GA and GP systems (e.g., [Goldberg 1989; Koza 1992]) generally describe non-adaptive methods for the recombination of the population members. Recent years have seen a growing interest in genetic operators (recombination strategies) that are responsive to the needs of the population they act upon. A genetic operator can be thought of as a program. An operator program can be hand-coded (e.g., "Pick and exchange two random GP sub-trees.") or can be learned, as in this chapter.

This chapter describes a procedure for the automatic design and use of new genetic operators. These *SMART* operator programs are co-evolved with the main population of programs and learn to recombine the main population programs better than random genetic recombination. Through a series of experiments, this chapter will show several domain independent features of the SMART operators' superiority to random recombination.

It seems appropriate to state here what this chapter is not. No attempt will be made in this chapter to describe the larger context and workings of the system in which these SMART operators are situated (see section 3.3). Further details on this system (beyond the language representation and SMART operators discussed in this chapter) can be seen in such works as [Teller and Veloso 1995a, 1995b]. In addition, this chapter is only an introduction to the SMART operator paradigm. Many questions of detail will not be answered, some for lack of space, and others because specific details and explanations are still under investigation in our current research.

This chapter can only briefly introduce the representation, the language, and some of the experiments. We hope that the domain independent operator features demonstrated by this chapter will contribute to genetic programming by inspiring future research into SMART operator programs and the general evolvability of the mechanics of evolution.

3.2 Background

This chapter will concentrate on the application of a population of co-evolved genetic recombination operator programs to the main population of programs they manipulate. Before the details of this process are explained, background information on similar approaches or goals will be outlined. The most important areas of work related to this chapter's focus are in the language representation that the main population programs (and SMART operator programs) are written in, and alternate techniques for dynamic recombination paradigms.

PADO's language representation, introduced in the next section, is structured not as a tree or DAG, but as an arbitrary graph with directed edges. At least graphically, a PADO program is reminiscent of Turing machines and Finite State Automata. Our indebtedness to the idea of graphs as representation for programs is wider and deeper than we can cite here. In evolutionary computation, the technique called evolutionary programming (EP) has been used to train graph structures like finite state machines and neural networks (e.g., [Saravanan and Fogel 1994]).

One work worth mentioning as an example is Karl Simms' work on evolving arbitrary graph structures that represent the morphologies of virtual animals. In [Simms 1994], for example, the question of how to "crossover" two graphs is addressed and dealt with in a simple and static way. Essentially, two graphs are randomly divided into two disjoint sets, one set from each graph is exchanged, and the "dangling" arcs are randomly reconnected. In section 3.4 we will return to this issue in search of more intelligent solutions.

The idea of evolving evolvability is not new. [Altenberg 1994] contains a summary of some past work and current ideas in this area. The concept of Adaptive Representations has been in the GA field for years. Adaptive representations encompasses all dynamic changes to the environment and the rules which govern how an evolving population grows and changes. Both in GA (e.g., [Julstrom 1995]) and in GP (e.g., [Rosca and Ballard 1995]) current research is investigating what can be improved about an evolutionary system by augmenting the environment to change probabilities, encapsulate evolved structures, and choose among existing operators. In chapters 4 and 5 of this book, Angeline and Iba respectively discuss at length very different approaches in the same field of adaptive operators.

The idea of learned programs that change (and hopefully improve) learned programs is not new to this chapter. For example, Schmidhuber describes how one might in principle learn to do this in a recurrent Neural Network [Schmidhuber 1993]. In [Schmidhuber 1987] Schmidhuber describes how this might be done with more traditionally structured programs, where each group of programs learns how to helpfully change the group of programs below it in a chain of program groups that is potentially without end.

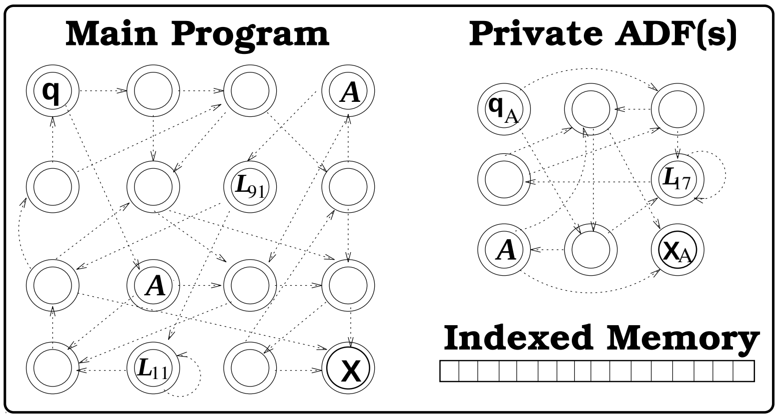

Figure 3.1
The general structure of a PADO program.

3.3 The Language

PADO (**P**arallel **A**lgorithm **D**iscovery and **O**rchestration) is a new learning architecture specifically designed for signal understanding. PADO's learning core is Genetic Programming. Figure 3.1 sketches the structure of a PADO program. Each program has three main components: a main program (referred to as MAIN), one or more private ADF programs [Koza 1994], and an indexed memory [Teller 1994a]. The MAIN program may call any of the private ADF programs it owns or any of the publicly available Library programs (see below). The private ADF programs may call each other, themselves, or any of the Library programs. The Library programs may, in turn, call any of the other Library programs, themselves, or any of the private ADF programs of the current PADO program. Any of these programs may access the indexed memory of the current running PADO program.

When a PADO MAIN program is run, a timer is started and the program is forcibly terminated when a pre-set time threshold is reached. This forced termination ensures that every program will halt with some answer (see section 3.3.1) in a limited amount of time.

The indexed memory is an array of integers indexed by the integers. Each program has the ability to access any element of its memory, either to read from it (READ *Index*) or to write to it (WRITE *NewValue Index*). This memory scheme, in conjunction with loop or recursive constructs in GP has been shown to be Turing complete [Teller 1994b]. Indexed memory can be seen as the simplest memory structure that can practically support all other memory structures. Indeed, indexed memory has been successfully used to build up complex data structures and mental models of local geography [Langdon 1995, 1996;

Teller 1994a].

The *ADF* programs (e.g., A in Figure 3.1) associated with each MAIN program are similar to standard GP ADF's (automatically defined functions). However, PADO ADF programs do not take a specific number of arguments but evolve to use what they need from the incoming argument stack (see below). In addition, they have internal loops and recursion. ADF programs evolve along with the MAIN program (as Koza does in [Koza 1994]).

The *Library* programs (e.g., L_{91} in Figure 3.1) are globally available programs (public ADFs) that can be executed at any time and from anywhere just like the ADF programs. But unlike the ADF programs, where each ADF may be run only during the execution of the PADO program of which it is a part, the *Library* programs are publicly available to the entire population. These *Library* programs are gathered from the ADFs of the most fit main population programs and survive through a competition for usage from highly fit main population programs. Details can be seen in [Teller and Veloso 1995a, 1995c]. While the creation, destruction, and global availability of these *Library* programs is different from the "module" concept, the maintenance of a pool of encapsulations of code is not [Angeline and Pollack 1993].

3.3.1 Looking Inside a PADO Program

Each PADO program is constructed as an arbitrary directed graph of nodes. As an arbitrary directed graph of N nodes, each node can have as many as N outgoing *arcs*. These arcs indicate possible flows of control in the program. In a PADO program each node has two main parts: an *action* and a *branch-decision*. Each program has an *argument* stack (see section 3.2). All PADO actions *pop* their inputs from this argument stack and *push* their result back onto the argument stack. These actions are the equivalent of GP's terminals and non-terminals. For example, the action "6" simply pushes 6 onto the argument stack. The action "Write" pops *arg1* and *arg2* off the stack and writes *arg1* into Memory[*arg2*] after pushing Memory[*arg2*] onto the argument stack.

Evaluating a GP tree is effectively a post-order traversal of the tree. This traversal requires an argument stack which is taken care of, in most traditional GP implementations, by the activation records stack for recursive function calls. Because there are many arcs coming into a particular node in the PADO language, we evaluate a part of the graph (indeed, the whole graph) as a *chronological*, not structural, post-order traversal of the graph. In other words, to see where the arguments to a particular node come from in PADO, we have to look to the previous nodes *in time* rather than the previous nodes *in the structure* (as per standard GP).

In stack-based GP the programs are written in Postfix Notation instead of tree form [Keith and Martin 1994; Perkis 1994]. If you maintain arity constraints through crossover (so that no function is executed before its arguments have been computed) then stack-GP is tree-GP. In contrast, If PADO's argument stack is empty when an argument request comes, a 0 is returned. So arity requirements are not considered in the construction or recombination of PADO programs. We mention these details because the argument stack mentioned in this section is not in itself a departure from GP, but, like stack-based GP, is simply an easier way of telling the same story.

After the action at node i is executed, an arc is taken to a new node. The branch-decision function at the current node makes this decision. Each node has its own branch-decision function that may use the stack top, the action type (e.g., "constant", "ADD", etc.) of the previously executed node, the memory, and constants to pick an arc. The next section will provide further branch-decision details.

Several special nodes are shown in Figure 3.1. Node q is the start node. It is always the first node to be executed when a program begins. Node X is the stop node. When this node is reached, its action is executed and then the program halts. When a program halts or is halted at the time-threshold, its response is considered to be the current value residing in some particular memory location (e.g., response = Memory[0]). If a program halts sooner than a pre-set time threshold, it is started again at its start node (without erasing its memory or stack) to give it a chance to revise its confidence value. Node A executes the private *ADF* program (starting at q_a) as its action. When (and if) the ADF reaches its stop node (X_A), control is returned to the calling program node that then executes its branch-decision function as normal. Node L_{91} executes Library program number 91 in a similar manner.

For the purposes of this chapter the programs' nodes have been limited to two outgoing arcs each. It is still possible for any node to have as many as N incoming arcs. This outgoing arc limit was fixed largely to make the special actions of the SMART operators (section 3.4.3) easier to understand. Though the following will not be justified here, we claim that there is a qualitative difference between programs with a maximum fanout of one (already a superset of GP-trees) and programs with a maximum fanout of two. Further, we claim that there is only a quantitative difference between programs with a maximum fanout of two and programs with higher maximum fanouts.

3.3.2 Looking Inside a PADO Node

The non-zero arity actions (non-terminals) for both main population and SMART operator programs are: ADD, SUB, MULT, DIV, READ, WRITE, ADF, IF-THEN-ELSE, PIFTE, MAX, MIN, LESS-THAN, EQUAL, NOT.

MAX and MIN double as OR and AND. All non-zero numbers are treated as TRUE and 0 as FALSE. PIFTE is a probabilistic IF-THEN-ELSE that takes 3 arguments. A random

Table 3.1
Branch-decision function templates

$(\text{Val}_1 > 0) \vee (\text{Val}_2 > 0)$	$(\text{Val}_1 > 0) \wedge (\text{Val}_2 > 0)$	$(\text{RandomVal} > \text{Val}_1)$	$(\text{RandomVal} > \text{Val}_2)$
$(\text{Val}_1 = 0) \wedge (\text{Val}_2 = 0)$	$(\text{Val}_1 = 0) \vee (\text{Val}_2 = 0)$	$(\text{Val}_1 > \text{Val}_2)$	$(\text{Val}_1 < \text{Val}_2)$
$(\text{Val}_1 = \text{Val}_2)$	$(\text{Val}_1 \neq \text{Val}_2)$	$(\text{Memory}[\text{Val}_1] > 0)$	$(\text{Memory}[\text{Val}_2] > 0)$

number is chosen and if it is less than *arg1* then *arg2* is returned, else *arg3* is returned. "Returned" in the PADO context is equivalent to "pushed onto the argument stack." "Takes X arguments" in the PADO context means "pops X elements off the argument stack to use as input." In this chapter, the main population programs were given access to Library programs through a *LIBRARY[i]* action. In addition, there are six signal-access actions that the PADO programs discussed in this chapter were given. These functions are {POINT, AVERAGE, MOST, LEAST, VARIANCE, DIFFERENCE}. More detail on them can be seen in [Teller and Veloso 1995a, 1995b].

The branch-decision function at a specific node is one of a preset set of function templates (table 3.1). Val_1 and Val_2 (table 3.1) can be any two of these three things: the action type of the temporally previous node (e.g. "WRITE"), the result of the action of the current node, and the node's branch-decision constant. The branch-decision function compares these two values through one of the twelve choices in table 3.1, and (for this chapter) follows ARC_1 on FALSE and ARC_2 on TRUE.

3.4 Intelligent Recombination

The basic function of a genetic recombination operator is to take K (usually two) programs as input and to produce some (usually K) new programs as output that replace the input programs in the population from which they were drawn. The context of the PADO representation lead to the need for recombination (crossover) of two arbitrary graphs. Partly because the "right" way to do graph recombination is not obvious, learning how to do graph recombination intelligently is a natural desire.

The SMART operators are *programs* that learn to do this graph crossover better than recombination operators acting at random. These SMART recombination operators co-evolve with the main population. Like the programs in the main population, the programs in the SMART operator population may begin as randomly generated programs. The programs in this SMART operator population are tested by allowing them to actually perform the recombinations on the main population. Their fitness values are a function of the relative fitness of the programs they take as input and the fitness of programs they produce as output. Our current research is studying ways to co-evolve the complete recombination paradigm. For this chapter, however, the recombination paradigm was fixed as shown below.

There is an important distinction between intelligent recombination in a particular recombination paradigm and an intelligent recombination paradigm. For example, in GP the standard recombination paradigm is crossover. A particular crossover between two parent programs requires *choosing* two subtree roots to exchange. The selection of these two roots can be done intelligently [Teller and Veloso 1995c] or randomly. There is an entirely different issue of whether the entire method of recombination (in this example crossover) is optimal. Part of our ongoing research effort is the more ambitious goal of learning the whole paradigm of recombination. Even if we constrain ourselves to a fixed recombination paradigm there is still room for smartness in this new representation. The paradigm for recombination on which this chapter will focus (as was used in the experiments) is as follows.

1. Take two programs.

2. Divide each into two node sets (fragments).

(a) Call the program fragments P1,P2,T1,T2 (where the programs were (P1-P2) and (T1-T2) before the recombination and will be (P1-T2) and (T1-P2) afterwards.)

(b) Label all arcs *internal* if they point to another node in the same fragment, or *external* otherwise.

(c) Label nodes in each fragment as *output* if they are the source of an external arc and *input* if they are the destination of an external arc.

3. Exchange one fragment from each program.

4. Recombine so that all *external* arcs in each fragment point to randomly selected input nodes in the new other fragment.

We could evolve both how to pick and how to reconnect the fragments. For this chapter, how they are reconnected is fixed as described above. In this constrained example, the only room for "smartness" in a recombination operator is, given two programs, to choose the fragments of those two programs that are likely to lead to highly fit new programs. Section 3.5 will show that we can co-evolve a subset selection strategy that is considerably better than random. The following subsections will provide necessary detail on the co-evolution of the SMART operators and the recombination strategy they will be compared against.

3.4.1 SMART Operator Co-evolution Mechanics

The SMART operator population exists in two demes. A deme (the use of which is also called the "island model") is a section of a population whose members intra-breed with much higher frequency than they inter-breed with individuals from other demes [Collins 1992]. One deme is responsible for recombining the ADFs and the other for recombining the MAIN programs. Each generation included a single individual drift between these two demes. Each deme was of size 150 for a total SMART operator population of 300 individuals. Each SMART operator program was allowed access to one ADF and no library programs.

$$R = \frac{\text{MainPopSize}}{\text{OperatorPopSize}} * \text{RecombinationPecent} \tag{3.1}$$

This chapter uses a simple application of the SMART operator population to the main population (see section 3.6.2 for alternatives). Each operator gets R recombination attempts. Since the MainPopSize was 2800 for the experiments in this chapter, and the RecombinationPercent was 85% this means that each of the SMART operators in the ADF deme take eight main population programs (four sets of two), recombine their ADFs, and insert eight new programs back into the main population. Each of the SMART operators in the MAIN deme take eight main population programs, recombine their MAIN programs, and insert eight new programs back into the main population. Here is an overview of the co-evolution process with SMART operators.

1. Measure approximate fitness of each main population program.

2. Measure approximate fitness of each SMART operator program.

3. Perform Fitness Proportionate Reproduction on main population.

4. Perform Sexual Recombination on main population using SMART operators.

5. Perform Fitness Proportionate Reproduction on SMART operator population.

6. Perform Sexual Recombination on SMART operator population.

The fitness estimate in step 2 is based on current and previous main population fitness values (see section 3.4.5 for exactly how this is measured). Step 6 is accomplished by using RANDOM recombination operators. See section 3.6.1 for alternatives. This process is co-evolution because the fitness of programs in each population (main and SMART operator) has an effect on the current (main $\rightarrow$ SMART) or future (SMART $\rightarrow$ main) fitnesses of the other population. The SMART operator population can start with random or seeded individuals (see section 3.6.3).

3.4.2 The Straw-Man

In order to assess the merits of the SMART operator programs and general strategy of SMART operator application and co-evolution we need a base line with which to compare them. This base line is, naturally enough, the kind of recombination operator a researcher might write: a *random* recombination operator. Random recombination in this representation, *within the general recombination paradigm outlined above*, chooses two random subsets of random sizes from the two input programs, and then exchanges them as outlined above. A SMART recombination operator, *by examining the input programs*, may choose two subsets of the two input programs that, when exchanged, are more likely to produce high fitness offspring.

This base-line random recombination operator has been improved since we first wrote it because of things we learned by studying highly fit SMART operators. We noticed that the distribution of sizes of program subsets to be exchanged was much wider and flatter for the SMART recombination operators than for the random recombination operator we originally wrote. When we went back to the random recombination operator code we noticed that to pick a random set of a random size it would place each node IN or OUT of the set (for each of the two programs) with probability 50%. This gives a binomial distribution of sizes of exchange sets. This was not a coding mistake and is not even in hindsight an obviously bad idea. Never-the-less, when we changed the random recombination operator code to pick a number between 1 and *ProgramSize*-1 and then pick that number of nodes at random to put in the set of nodes to exchange, the "fitness" (as defined for the SMART operators, section 3.4.5) of the random recombination operator rose dramatically. It is exactly because these strategies are obvious only in hindsight that we should learn, not design by hand, our recombination strategies.

3.4.3 SMART Recombination Primitive Actions

The SMART recombination programs need the ability to examine their input programs in sufficient detail to make an informed decision. The SMART recombination programs need to be able to indicate their specific decisions. For SMART recombination operator programs this specification should take the form of "Here are the subsets of the two input programs to be exchanged." Like the main population programs, the SMART recombination programs are given the standard PADO actions (see section 3.3.2). Table 3.2 shows the SMART operator special actions.

When a SMART recombination program is run, it continues until it executes special action 13 or the time-threshold is reached, whichever comes first. Whichever does occur, the program is halted and the two sets (SET_0 and SET_1) the SMART recombination operator

Table 3.2
The special actions given to the SMART recombination programs.

SA#	**S**pecial **A**ction Description
SA-1	Pick at random a new CurrentNode$_V$ $(S/\overline{S})$
SA-2	Pick a specific node to be the new CurrentNode$_V$
SA-3	Pick node pointed to by ARC$_I$ of CurrentNode$_V$ to be the new CurrentNode$_V$
SA-4	Add CurrentNode$_V$ to SET$_V$
SA-5	Add Children of CurrentNode$_V$ to SET$_V$
SA-6	Delete CurrentNode$_V$ from SET$_V$
SA-7	Delete Children of CurrentNode$_V$ from SET$_V$
SA-8	Make SET$_V$ be a random set from Program$_V$
SA-9	Return Size of SET$_V$
SA-10	Return Size of Program$_V$
SA-11	Return Number of Children of $(S/\overline{S})$ SET$_V$ that are in $(S/\overline{S})$ SET$_V$
SA-12	Return a *Value* of CurrentNode$_V$
SA-13	DONE

V → Program$_0$ or Program$_1$ as specified by an argument
Value → Action type or Branch-decision type or ARC value as specified by an argument
$(S/\overline{S})$ → domain of action is SET or $\overline{\text{SET}}$ as specified by an argument.

was responsible for building up are examined. If both sets have at least 1 and no more than ProgamSize-1 elements in them (i.e., if the exchange will be a meaningful recombination) then the recombination is performed (see beginning of section 3.4). Otherwise, as will be discussed in section 3.4.5, random recombination is performed on the two input programs and the SMART recombination operator receives 0 fitness for that trial.

One of the most important special actions is special action eight. This action sets SET$_i$ to a random sized subset of randomly selected nodes from Program$_i$ where i is specified (to be 0 or 1) by the single argument this action takes. In short, this is the random operator that the SMART operators are competing against. This is exactly why the SMART operator paradigm is safe to incorporate into the standard GP tool box: this primitive has to be written anyway (since it is the random operator the system uses). In addition, if we improve the random operator and install this improved version in our system, then the SMART operators, as well as the random operator, will reap the benefits.

Of course, if this were the only special action primitive there would be nothing to lose from the SMART operators, but also nothing to gain. If, however, the SMART operators are given the power either to build up sets from scratch, or to purposefully alter randomized node sets they had requested (from SA-8), then there is a potential for general operator improvement. While we believe that all of the special actions listed in table 3.2 give this potential power, SA-5 and SA-7 are of particular interest. Programming a depth-first

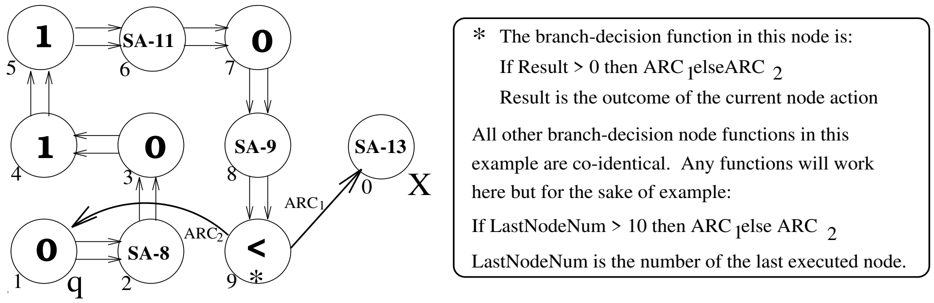

Figure 3.2
An extremely simplified SMART operator program example.

traversal in a graph is non-trivial. SA-5 and SA-7 make such traversals possible (for the special cases of adding and deleting set nodes) through repeated applications of a single action. A few well thought out special actions can make a difficult computation easy.

3.4.4 SMART Operator Example

A trivial SMART operator will be explained here as a foothold for the imagination. It is important to understand that the example pictured in Figure 3.2 is smaller than a typical SMART recombination operator by almost two orders of magnitude. In addition this example program is readable and concise in a way that never happens under normal evolution conditions. To begin to understand the SMART recombination example pictured in Figure 3.2, let us start by writing its operation in English pseudo-code:

1. Set SET_0 to a random subset of nodes from $Program_0$. (Nodes 1 and 2)
2. Get the number of arcs whose source *and* destination is SET_0. (Nodes 3,4,5 and 6)
3. Get the number of nodes in SET_0. (Nodes 7 and 8)
4. if (result of 3 $<$ result of 2) then STOP, otherwise goto step 1. (Node 9)

The next question is, what does this really do? The answer is:

> Keep picking random subset of $Program_0$ for SET_0 until more than half the children of nodes in SET_0 are in SET_0. In other words, keeping picking subsets for SET_0 until it passes the 50% intra-connectivity threshold (i.e. the point at which more than half the outgoing arcs from nodes in the set point back to nodes in the set).

The branch-decision function can, in general, be different for each node. For the sake of simplicity, Nodes 1 through 8 all have the same function (see Figure 3.2). This function, for all eight nodes, will always pick ARC_2 as the control transfer arc. Neither the number of arcs, where they point, nor the branch-decision function need exhibit this kind of regularity. Now here is a blow-by-blow account of each of the ten nodes and what they actually do:

Node 0: This is the stop node. Generally, when PADO reaches this node, the PADO program's current state is recorded and then, providing the time-threshold has not yet been reached, the program is restated at node 1 (q). In this case, the stop node happens to executes Special Action 13 (SA-13), which signals that the SMART operator has made its choice and need not be restarted.

Node 1: This is the start node. It places a 0 on the program's argument stack.

Node 2: This node executes Special Action 8 (SA-8). This action takes one argument V and sets the SMART recombination operator's SET_V to a random subset of nodes in Program_V. In this case V will be 0 since that is what is on top of the argument stack.

Node 3: This node places a 0 on the argument stack.

Node 4: This node places a 1 on the argument stack.

Node 5: This node places a 1 on the argument stack.

Node 6: This node executes SA-11. SA-11 takes three parameters (Z, Y, V) $(V > 0 \rightarrow V = 1)$ and returns the number of arcs whose source node is *in* SET_V (if $Y > 0$) or *out* of SET_V (if $Y = 0$) and whose destination node is *in* SET_V (if $Z > 0$) or *out* of SET_V (if $Z = 0$). Generally, this action can measure the intra- or inter-connectivity of SET or $\overline{\text{SET}}$. In this case Y and Z are 1 and V is 0, so the effect of this action is to place on the argument stack the number of arcs with source nodes *in* SET_0 and destination nodes *in* SET_0.

Node 7: This node places a 0 on the argument stack.

Node 8: This node executes SA-9. SA-9 takes one parameter (V) and returns the number of nodes in SET_V. In this case, 0 is on the top of the argument stack, so this action puts the number of nodes in SET_0 onto the argument stack.

Node 9: This node executes the standard PADO action "less-than", which takes the top two values off the argument stack (X, Y) and puts a 1 on the argument stack if X is less than Y and a 0 otherwise. In this case, the top two values on the argument stack are the results of the actions from nodes 6 and 8. This node's branch-decision function is "If RESULT $>$ 0 then ARC_1 else ARC_2" (see Figure 3.2). RESULT is the result of this node's action.

When this SMART operator program finishes, the environment will take SET_0 (the Program_0 nodes to be exchanged) and SET_1 (the Program_1 nodes to be exchanged, left empty in this example) and exchange and recombine them as described in section 3.4.

Table 3.3
Hypothetical Recombinations

ParentFit1 = 0600 and ParentFit2 = 1000 → ChildFit1 = 0550 and ChildFit2 = 0550	(1)
ParentFit1 = 0600 and ParentFit2 = 1000 → ChildFit1 = 1100 and ChildFit2 = 0000	(2)
ParentFit1 = 0100 and ParentFit2 = 1000 → ChildFit1 = 0500 and ChildFit2 = 1000	(3)
ParentFit1 = 0100 and ParentFit2 = 1000 → ChildFit1 = 0100 and ChildFit2 = 2000	(4)

3.4.5 SMART Operator Fitness

In any evolutionary computation, the fitness measure is largely responsible for directing the system's search. The fitness measure embodies any explicit goals of the system and largely determines the kind of behaviors that are produced. In a qualitative way we all know what we mean by a "good" operator and a "bad" one. What we need however is a quantitative measure for the fitness to drive our SMART operator populations.

The goal of the system with respect to the main population is to maximize the fitness of the maximally fit individual (or occasionally top few individuals) in the main population. At least while a perfect solution has not been reached most researchers assume that there is a strong correlation between the rise in average population fitness and maximum population fitness. Given this scant knowledge, what should a fitness measure of operators be?

The application of SMART operator programs to a main population of programs is not dependent on the particular choice of fitness measure, as long as that fitness measure is really an approximation to what we mean by a "good" operator. Let us propose one metric which is simple to understand and to measure. It is given here not to end the discussion on this subject but to ground the experiments and discussions in the rest of this chapter.

In any kind of recombination, the operator takes one or more programs out of the main population, changes them, and replaces them in the population. The fitness of the operator should be a function of the behavior of each input program and the output program that replaced it in the main population. In addition, the mechanism of evolutionary computation is such that when a program has low fitness, we do not care exactly how bad it is. We care when a program has high fitness relative to the population fitness distribution.

The main justification for the fitness measure below is that we want to produce programs that are strictly more fit than their parents at least some of the time. In case 1 of table 3.3 the recombination leads to the same ratio of child to parent fitness as in case 2. However since our goal is really to maximize the child-parent fitness ratio only for fitness improvements, case 2 is probably the one that should return higher fitness to the operator that created it.

In examples 3 and 4, we see that both replacement pairs in the main population include one improvement. In this extreme comparison the fitness ratio is five for the Parent1 → Child1 operator action in case 3 and is a fitness ratio of two for the Parent2 $\longrightarrow$ Child2 operator action in case 4. Even in this situation we would like the operator that caused these changes to be rewarded higher for case 4 than for case 3 because the total fitness increase is larger. The desire to weight these cases as described is why MaxFitness (M below) is added to the numerator and denominator when the fitness ratios are summed.

Let P_V be *the fitness of some program V to be recombined.*

Let C_V be *the fitness of the program output to replace V in the main population.*

Let $\mathcal{S}$ be *the set of all trials on a particular generation for some operator X such that $C_V > P_V$. $|\mathcal{S}|$ is the number of programs output by operator X whose fitness is strictly greater than the fitness of the input program it is replacing in the population.*

Let $\mathcal{T}$ be *the number of main population programs given to operator X to recombine.*

Let M be *the maximum fitness a main population program can have.*

And define

$$\Omega = \sum_{i \in \mathcal{S}} \frac{C_i + M}{P_i + M} - 1 \tag{3.2}$$

Given these definitions let us define the fitness of an operator (F_o) as:

$$F_o = \frac{|\mathcal{S}|}{\mathcal{T}} * \Omega \tag{3.3}$$

In English, the fitness of an operator translates to:

> The percentage of the time that the main population program the SMART operator program places back in the main population is more fit than the main population program it is replacing (this is the $\frac{|\mathcal{S}|}{\mathcal{T}}$ part of equation 3.3) *TIMES* a fitness measure (relative to M) of how much more fit the improved programs are than the programs they replace (this is the Ω part of equation 3.3).

As a final note about operator fitness, the question arises, "What fitness does a SMART operator receive that does not split its input programs into two sets each?" Such an operator reverts to the random operator, so that the main population is not punished for the failing of that SMART operator (more on that subject in section 3.6). The answer, at least for the experiments and discussions in this chapter, is that such a SMART operator receives no fitness boost, even if the random operator to which it has reverted to performs well. The rationale for this strategy is that there is a period of "dumbness" (the first few generations) that the SMART operators have to go through to reach more intelligent recombination strategies. There is less pressure on the SMART operators to learn during this period if the random operator, acting as back up for the inept SMART operators, provides these inept SMART operators with non-zero fitness.

3.5 Experiments

The most obvious aspect of the graphs in this section is the change in SMART operator performance (due to evolution) as the generations pass. However, the main population is also changing as these generations pass, and this causes much of the changes in the graphs. While these changes (the general dynamics of the main population at generation Y) are hard to show on the graph, the relevant aspect for this chapter can largely be understood by studying the random operator curves. Since the random recombination operator remains unchanged in its operation as the generations pass, the changes to its performance curve are entirely attributable to the changing main population dynamics. Keeping this in mind will help in deciphering this section's graphed results.

Experiment-specific details will be given with the experimental results, but several pieces of experimental methodology were fixed for this entire chapter. Figures 3.4, 3.5, and 3.6 in sections 3.5.2, 3.5.3, and 3.5.4 all show data taken from 5 runs from each of 3 domains: The three domains are the sound domain mentioned in section 3.5.2 [Teller 1995b], a robotic image object recognition task [Teller and Veloso 1995a], and a difficult manufactured image classification problem [Teller and Veloso 1995d].

For the main population, all experiments used 85% recombination (SMART or RANDOM), 5% random mutation, and 10% simple reproduction. The SMART operator population was subjected to 40% random recombination, 10% random mutation (see section 3.6), and 50% simple reproduction. The main population consisted of 2800 programs. Each one was allowed one ADF and access to any of the 150 Library programs (public ADF's). The node size limit on all programs (MAIN, ADF, and library) in both the main and SMART operator populations was set to 256. The indexed memory size was set to 256 elements for both the main and SMART operator programs. The time-threshold for all programs (main population and SMART operators) was approximately equal to 8000 node evaluations. A typical run to generation 100 in PADO took about a day on a DECstation 5000/20.

The length of this chapter does not provide for a detailed study of the SMART mutation as well as the SMART recombination operators. The reason that we concentrate on the SMART recombination operators is that mutation in GP is primarily used to maintain diversity. Random mutation does this job quite well. However, discussion on SMART mutation operators can be found in [Teller 1996].

3.5.1 System Performance

Figure 3.3 shows an example of the SMART operators' abilities. This graph shows a sound classification experiment using sounds extracted from the SPIB database (ftp spib.rice.edu). There are seven sound classes (Factory-noises-1, M109-engine, Buccaneer-

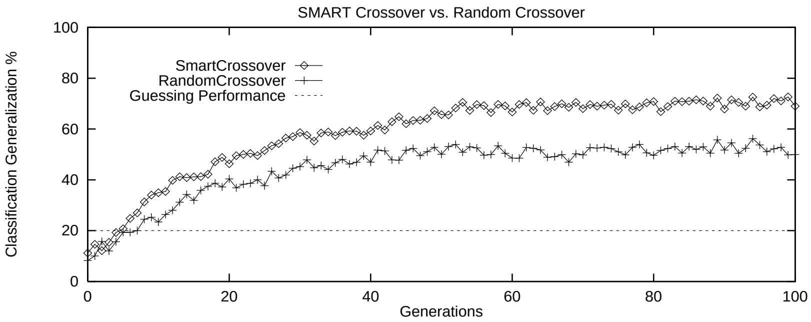

Figure 3.3
PADO System Sound Classification Generalization Percentage on Test Sounds.

jet-engine, Machinegun, Volvo-engine, Canteen-babble, Factory-noises-2). PADO was trained on 35 examples of each sound. The graph in figure 3.3 shows the generalization ability of the orchestrated PADO system (essentially an average of the best few programs in the population [Teller and Veloso 1995a]) on a set of unseen test examples over generations (averaged over 10 runs). For more details of this problem, see [Teller 1995b].

This graph is a positive, representative example of the effect SMART operator program application has on performance in PADO. This chapter could detail more about the main population performance benefits of SMART operator evolution and application only at the expense of detail about the SMART operators themselves. Because this chapter is primarily about the co-evolution of intelligent recombination operators, and not about the PADO classification learning system, the further performance benefits explicitly stated in this chapter will be confined to the following statement.

> In our experience, the use of co-evolved SMART operator programs to aid evolution has almost always helped performance, sometimes dramatically, and has never in our experience been a noticeable hindrance in either speed or population performance.

On average, the evolution and application of the SMART operators takes about 30 seconds per generation. That means that a run to generation 100 spends 3.4% of its time on the SMART operators. This is a negligible price to pay for more intelligent recombination.

3.5.2 Domain Independent Operator Performance

The two graphs in figure 3.4 show the percentage of *real improvement* for the SMART and RANDOM recombination operators across 3 domains. Real improvement can generally

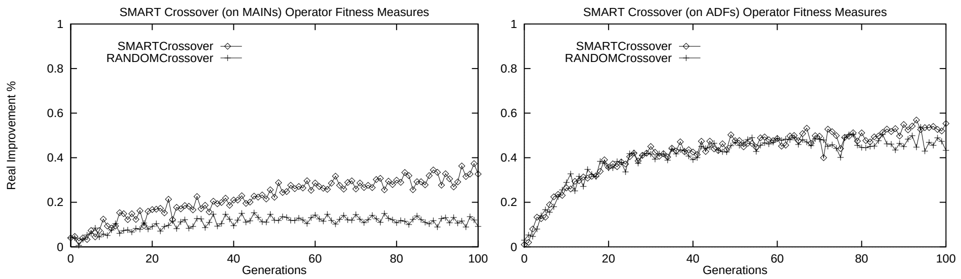

Figure 3.4
Tracking recombination-operator real improvement percentage across generations (in MAIN and ADF graphs).

be thought of as an instance where a main population program is measured to be more fit after recombination than before. In all three of the domains over which these graphs were averaged, the fitness values (for main population programs) ranged from -MaxFitness to +MaxFitness where a fitness of 0 could be achieved by random guessing on the part of the programs. In such a case, real improvement caused by a SMART operator can more accurately be thought of as instances in which a program is measured to be more fit after recombination than before and in which its new fitness is greater than 0. Figure 3.4 shows the percentage of the time that this happened.

Figure 3.4 shows percentage real improvements, not operator fitness as defined in section 3.4.5. The reason for this is two-fold. Because these graphs are averaged across 3 domains, both the absolute and relative changes in main population fitness (two of the factors in operator fitness) vary widely between domains. These varied absolute and relative fitnesses tend to make the graphs less readable and less meaningful without detailed information about the absolute and relative fitness dynamics in each of the domains.

These graphs are averaged over five runs in each of the three domains, all of which used SMART operators as the recombination strategy. In each run the average SMART operator real improvement was taken from the actual recombination phase. The random recombination operator was applied to the same population on each generation and its average improvement measured on this temporary new population. Then, this random operator-generated main population was discarded and the next population began.

The question is "Can SMART recombination operators learn (co-evolve) to produce a higher percentage of *real improvements* per recombination than the random recombination strategy over a range of domains?" If they can, and we can show it, we will have taken an important step towards showing the general usefulness of the SMART operator paradigm.

As can be seen from the MAIN graph in figure 3.4, the power of the SMART operators seems to come from their increased ability to produce improvements when performing

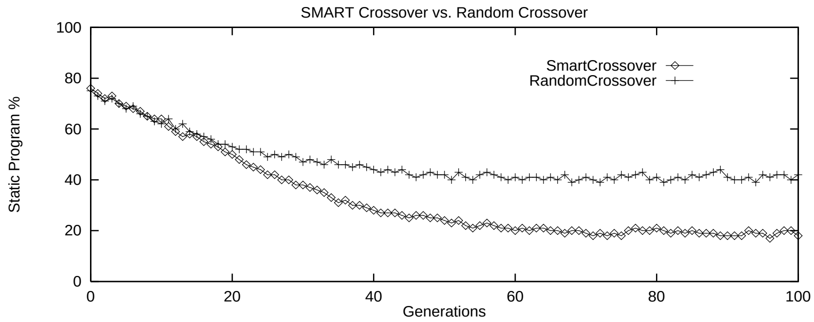

Figure 3.5
Percent of the Population that gives a static response

recombination in the MAIN program. This may be because there are strategies for good recombination which are simply easier to learn for the MAIN program graphs than for the ADF program graphs. Interestingly, both SMART and RANDOM recombination are considerably more likely to improve a program by performing recombination on its ADF than on its MAIN program. While it is tempting to guess that this higher real improvement percentage happens because most ADFs are simply unused and a program's fitness may simply be measured higher next generation, this turns out not to be the case. In these runs, about 80% of the main population programs actually used their ADF. Figure 3.4, then, suggests that the rate (or even maximum height) of evolution might be raised by doing recombination more often on the ADFs than on the MAIN program graphs. Since this was not the focus of research here, the experiments mentioned all performed recombination on the MAIN and ADF programs equally often.

3.5.3 Static Programs

A small subset of all programs are sensitive to their inputs (i.e., their output changes as their input changes). Most programs are not sensitive to their inputs. Clearly, these *static* programs will have low fitness and therefore are simply a drain on the evolutionary system. Any operator which can reduce the percentage of its output programs that have this undesirable aspect has a potentially (see below) desirable trait. The static program production difference shown in figure 3.5 is an artifact, a side-effect of the operator fitness measure. The notion of a static program is absent from the actual operator fitness calculation.

Like any measure of SMART operator performance, this measure is not perfect. It is possible that an operator exists such that nine times out of ten it produces a static individual and the tenth time it produces an extremely high fitness program. Another possible situation

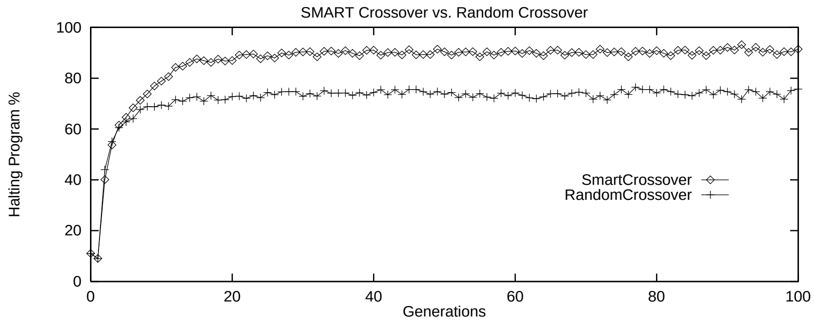

Figure 3.6
Percent of the Population that halts when recombination is applied

is a SMART recombination operator that always produces one static child and one very highly fit child. Such hypothetical operators aside, reducing the number of static programs created seems safely to be an advantage for a genetic programming system.

3.5.4 Halting Programs

As this chapter has described, PADO does not have to require that a MAIN program halt in order for that program to have non-zero fitness. By using a time-threshold and an anytime extraction procedure (e.g., answer = MEMORY[0]), we do not have to require that programs find an answer, know that they are done, and transition to their stop node. Indeed, when no self-halting criterion is enforced, by generation 100 only about 1% of the programs transition to their stop node even once during a particular execution. So, there does not appear to be a selective advantage to stopping and being restarted during a run. The reason the stop node exists in the MAIN programs is that an experimental version of PADO is beginning to use MAIN programs (as well as ADF programs) in the Library. This requires that MAIN programs be able to relinquish control.

As an experiment, however, we pressured programs to halt by assigning zero fitness to programs that had not self-halted by the time-threshold. This situation presents an interesting problem to the recombination operators: the fitness of an individual will drop to zero if recombination causes the new program to no longer halt. In figure 3.6 we see that the SMART operators are able to significantly reduce the percentage of newly created programs that never halt. While this is not always of direct fitness advantage it is, in this experiment, another indication of the SMART operators' flexibility to respond to demands of the environment. The important thing to notice is that an interesting operator "sub-task" was partly solved here through the co-evolution of recombination programs.

3.6 Discussion

3.6.1 SMART Operator Recombination

One of the most obvious issues of SMART operator evolution is "Who is recombining the SMART operators?" As was mentioned in section 3.4, the results of this chapter have used the simplest solution: random recombination. Isn't this exactly what we are trying to avoid by having the SMART operators? Yes, at a meta-level. In general there are three solutions to this meta-recombination problem: random recombination, self-recombination, and Meta-SMART recombination.

Random recombination is a reasonable option. This chapter does not claim that random recombination in PADO (or GP in general) does not work. Probably the SMART operators would learn a little faster if the recombination used on them was more intelligent but there is a price.

Self-recombination seems at first like a good idea but we have found it (empirically) to be lackluster. The problem appears to be that if the SMART operators' fitness is based on their ability to recombine both the main population and each other then they are evolutionarily "pulled" in two different fitness directions. If their fitness is based only on the main population recombination results, then they are evolutionarily "pulled" away from the sort of intelligence needed to recombine each other. We hypothesize these results occur because the syntactic needs of the two populations, even on the same generation, are very different.

Meta-SMART recombination is the third possibility. Why not make Meta-SMART operators for the SMART operator population? This strategy is reminiscent of [Schmidhuber 1987]. In our opinion this simply shuffles the problem around without solving it; what happens to the Meta-SMART operators?

3.6.2 SMART Operator Application

This chapter has used the simplest application of the SMART operator population to the main population: each operator gets R recombination attempts (equation 3.1).

There are several other options, which offer potential improvements to the paradigm. First, let us review why this simple strategy should work at least as well (baring over-greediness) as using only random recombination. Any SMART operator that doesn't act gets zero fitness, but the programs that were input are randomly recombined anyway. So the inactive SMART operators do not pose a fitness disadvantage (relative to random operators) to the main population. All SMART operators are given the random operator as a safety net (SA-8). This means that it is trivial for SMART operators to do at least as well as the random operator would do. Depending on the representation and recombination styles for the SMART operators, only a small fraction of the population will do something and not use the safety net primitive. It is only this small fraction of the SMART operator population that stands a chance of bringing the population's total recombination performance down below that of the random recombination strategy.

Here is an example of another strategy: *conservative SMART operator reapplication*. FPR below stands for Fitness Proportionate Reproduction.

1. Save main population state
2. Perform Main population FPR and recombination using *all* SMART operators
3. Test the fitness of each program in the main population
4. Use relative main population fitnesses from 1 and 3 to get SMART operator fitnesses.
5. Perform FPR and recombination on the SMART operators
6. Return the main population to its saved step 1 state.
7. Perform Main population FPR and recombination using the K best SMART operators.
8. Test fitness of main population programs
9. Start Generation X+1. Goto step 1

Up to step 5 this process is exactly the strategy described in section 3.4. A natural question is, "In step 4, why not save the better individuals and integrate them with the main population in step 7?" The reason is that without very clever program difference-measures this strategy can quickly kill diversity.

This strategy has the advantage of only the most skillful SMART operators having a real effect on the main population's journey through fitness landscape. The cost for this is a slow down of a factor two. In our limited experience with conservative reapplication, this cost is often made up for either with an earlier generation at which the population reaches some particular level of fitness, or with the increased fitness maximum the population achieves.

3.6.3 Beyond Co-evolution

The simplest addition to the SMART operator paradigm is probably the concept of seeding each new population of SMART operators (at generation 0) with a population that has already evolved in the same or a similar domain. Surprisingly, even for the same domain, this has not worked well experimentally, though we have so far been unable to find a satisfactory theoretical explanation. In fact, even the most fit SMART operator from generation 100 is often less good on generation 25 (when seeded) than the most fit SMART operator that evolved there from scratch.

More generally, the question may be asked, "What happens when the co-evolution of the SMART operators is discarded completely?" In other words, it might be possible to evolve a set of SMART operators that would generalize across runs, across domains, or even across generations. Suppose that a set of SMART operators were co-evolved on domain X and that the population of SMART operators after generation Y was saved to a file called SMART.X.Y. To generalize across runs would mean that SMART operators (SMART.X.Y) could be used on each run in domain X and that on generation Y the SMART.X.Y population would be used for recombination. To generalize across domains would mean that SMART operators (SMART.X.Y) could be used on each run in any domain (not just domain X) but on generation Y the SMART.X.Y population would be used for recombination. To generalize across generations would mean that SMART operators (SMART.X.V) could be used on each run of domain X but on each generation Y, the SMART operator population SMART.X.V for some fixed V could be used. The term "could be used" in this paragraph means not only "was worth using because it generally outperformed the random operators," but also "was worth using because it generally outperformed the SMART operators that would have co-evolved to that generation." The summary of our experiences has been that while canned (i.e. saved and recovered) SMART operators generalize well across runs, they do not generalize well across domains or generations. In practice, we have found no seeding paradigm worth the effort. However, these results are little better than anecdotal; our on-going work in this area may still bear fruit.

3.6.4 Future Work

While this SMART operator paradigm has been used successfully on tree-structured programs [Teller and Veloso 1995c] there has not yet been a system using tree-structured functions (evolved programs with no loops, recursion, or memory) that has used SMART operators, successfully or otherwise. This leaves open the possibility that SMART operators may (or may not) work in a more constrained language space, such as tree-structured functions. The studies we have done do not tell us what tree-structured SMART operator functions could get done. The effect that representation and language expressiveness (of both the programs to be recombined and the SMART operators) has on the effectiveness of the SMART operator technique is an area requiring further investigation.

This chapter has already mentioned several of our current and future research directions. There are two directions, however, that stand out as most important and interesting to us. The first is a more thorough investigation of alternative SMART operator application paradigms. Two such alternatives that this chapter touched on were conservation

SMART operator re-application and fixed (non-co-evolved) SMART operator populations. These and other alternative approaches may eventually turn out to be superior to the basic application strategy outlined in this chapter.

The second avenue of research that we are actively pursuing is further theoretical analysis to improve our understanding of how SMART operators really work. Understanding the SMART operators and how they work can lead to two developments. First, this understanding can tell us how to improve the SMART operators and potentially give us general insights into the intelligent exploration of the space of algorithms. Second, learning what the SMART operators are doing may lead to improvements in the representation on which they operate. In other words, it may be the SMART operators who make the best suggestions (albeit subtly) about how to change the language representation.

3.7 Conclusions

This chapter has tried to convey three aspects of a paradigm: what it is, how it works, and an intuition for why it works. The real point of the chapter is to convince other researchers that co-evolution and application of SMART operator programs can be made more effective than human-designed operators with minimal extra effort.

The experiments in section 3.5 demonstrate the ability of co-evolved SMART operators to outperform random recombination operators in a variety of domains. These experiments show absolute performance and percentage real improvement gains due to the SMART operators. In addition, the experiments show two main population attributes, one bad and one good, that were reduced and increased respectively over random operator performance, even though the operator fitness measure took neither attribute explicitly into account.

This chapter is not intended to be a showcase for impressive results. Rather, we hope to have conveyed a strong intuition about why there is much to gain and almost nothing to lose by incorporating SMART operators into the general framework of genetic programming. There are numerous possible additions to and adaptations of the SMART operator paradigm, many of which were mentioned in this chapter. We hope this chapter is the first word, not the last, a primer and inspiration, not a bible or an answer, on SMART operator program evolution.

Acknowledgment

Manuela Veloso has been invaluable to me in my PADO work. P. Angeline, H. Iba, L. Spector, J. Rosca, E. Siegel, Z. Teller, and E. Teller all helped me by reviewing this chapter. I would also like to thank the Fannie and John Hertz Foundation for its kind support.

Bibliography

Altenberg, L. (1994) The Evolution of Evolvability in Genetic Programming, *Advances In Genetic Programming*, Kinnear, K. (ed.), MIT Press.

Angeline, P. and Pollack, J. (1993) Evolutionary Module acquisition, In *Proceedings of the Second Annual Conference on Evolutionary Programming*, D. Fogel (ed.), Evolutionary Programming Society. pp. 154-63

Collins, R. (1992) Phd Thesis: Studies in Artificial Evolution, University of California in LA, Department of Computer Science

Goldberg, D. (1989) *Genetic Algorithms: In search, optimization, and machine learning*, Addison-Wesley Press.

Julstrom, B. (1995) What Have You Done for Me Lately? Adaptive Operator Probabilities in a Steady-State Genetic Algorithm, In *Proceedings of 1995 International Conference on Genetic Algorithms*, Morgan Kauffman. pp. 81-7

Keith, M. and Martin, M. (1994) Genetic Programming in C++: Implementation Issues, In *Advances In Genetic Programming*, Kenneth E. Kinnear, Jr. (ed.), MIT Press.

Koza, J. (1992) Genetic Programming, MIT Press.

Koza, J. (1994) Genetic Programming II, MIT Press.

Langdon, W. (1995) Evolving Data Structures with Genetic Programming, In *Proceedings of the Sixth International Conference on Genetic Algorithms*, Stephanie Forrest (ed.), Morgan Kauffman. pp. 295-302

Langdon, W. (1996) Data Structures and Genetic Programming, In *Advances in Genetic Programming 2*, P. Angeline and K. Kinnear (eds.), MIT Press.

Perkis, T. (1994) Stack-Based Genetic Programming., In *Proceedings of the First IEEE International Conference on Evolutionary Computation*, IEEE Press. pp. 148-53

Rosca, J. and Ballard, D. (1995) Causality in Genetic Programming, In *Proceedings of 1995 International Conference on Genetic Algorithms*, Morgan Kauffman. pp. 256-63

Saravanan, N. and Fogel, D. (1994) Evolving neurocontrollers using evolutionary programming, *Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE Press. pp. 217-22

Schmidhuber, J., (1987) Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-...hook, In *Institut fur Informatik, Technische Universitat Munchen*

Schmidhuber, J., (1993) A neural network that embeds its own meta-levels, In *Proceedings of 1993 IEEE International Conference on Neural Networks*, IEEE Press. 407-12.

Simms, K. (1994) Evolving Virtual Creatures, In *Proceedings of the 21st International SIGGRAPH Conference*, ACM Press. pp. 15-22

Teller, A. (1994a), The Evolution of Mental Models, In *Advances In Genetic Programming*, Kinnear, K. (ed.), MIT Press.

Teller, A. (1994b) Turing Completeness in the Language of Genetic Programming with Indexed Memory., In *Proceedings of the First IEEE World Congress on Computational Intelligence*, IEEE Press. pp. 136-46

Teller, A. and Veloso, M. (1995a) PADO: A new learning architecture for object recognition, In *Symbolic Visual Learning*, Katsushi Ikeuchi and Manuela Veloso (eds.), Oxford University Press.

Teller, A. and Veloso, M. (1995b) Program Evolution for Data Mining, Sushil Louis (ed.), In *The International Journal of Expert Systems. Third Quarter. Special Issue on Genetic Algorithms and Knowledge Bases*, JAI Press.

Teller, A. and Veloso, M. (1995c) PADO: Learning Tree Structured Algorithms for Orchestration into an Object Recognition System, Technical Report number CMU-CS-95-101, Department of Computer Science, Carnegie Mellon University

Teller, A. and Veloso, M. (1995d) A Controlled Experiment: Evolution for Learning Difficult Image Classification, In *Seventh Portuguese Conference On Artificial Intelligence*, Springer-Verlag, pp. 165-76

Teller, A. (1996) Evolving Programmers: SMART Mutation, Currently unpublished technical report, Department of Computer Science, Carnegie Mellon University, Forthcoming