final version:
Prerational Intelligence,
Adaptive Behavior and Intelligent Systems Without Symbols and Logic,
Ritter, Cruse, Dean (Eds.),
Volume II, Studies In Cognitive Systems 36, Kluwer, 2000

previously as:
Conf.proc. "Integration of Elementary Functions into Complex Behavior"
Ritter, Cruse, Dean (Eds.),
Zentrum für interdisziplinäre Forschung (ZiF)
Bielefeld, Germany, July, 1994

Schemas and Genetic Programming

Andreas Birk * Wolfgang J. Paul †

Abstract

With the help of schemas and genetic programming we describe systems which

- interact with the real world
- make theories about the consequences of their actions and
- dynamically adjust inductive bias.

We present experimental data related to learning geometric concepts and moving a block in a microworld.

1 Introduction

To investigate the mechanisms which enable systems to learn is among the most challenging of research activities. In computer science alone it is pursued by at least three communities [Car90], [Nat91], [Rit91]. The overwhelming majority of all studies treats situations with strong *inductive bias*, i.e. there is a fairly narrow class H of algorithms and the concept or algorithm to be learned is known *a priori* to lie in that class H.

With the help of schemas in the sense of Drescher [Dre91] and genetic programming [Koz92] we will describe here systems which

1. interact with the real world via effectors and sensors,
2. make theories about the consequences of their actions and
3. dynamically adjust inductive bias.

A typical example is a system equipped with a robot arm and a camera which learns to 'play' with wooden blocks. The initial program of the system *does not refer to cameras, arms or blocks*. 'Playing' should involve looking at the blocks, rotating them and eventually building towers. We will report experimental results from a partial implementation.

The remainder of this paper is organized as follows. In section 2 we briefly review genetic programming. We point out some relations to the work of Piaget

*Vrije Universiteit Brussel, AI-Lab, Pleinlaan 2, 1050 Brussels, Belgium; cyrano@arti.vub.ac.be
†Universität des Saarlandes, Computer Science Department, Im Stadtwald, 66123 Saarbrücken, Germany; wjp@cs.uni-sb.de

[Pia91] and Occam's razor. In section 3 we briefly review the schema mechanism from Drescher [Dre91]. In sections 4, 5 and 6 we modify this mechanism in three crucial ways:

1. we change the mechanism by which schemas are chained
2. we treat a network of schemas as a *spatial model* of the world. This permits us to introduce a *standpoint*, which models the system's own place in the world and
3. we change the mechanisms for valuating knowledge.

In section 7 we report experimental results. The modified mechanism finds several networks of schemas which were not or only partially found by the original schema mechanism. We also report times required to discover basic geometric shapes with the help of genetic algorithms.

2 Genetic Programming

2.1 Definitions

Genetic Programming is a heuristic technique for searches in sets L of programs or numbers which are large and poorly structured. In order to apply it one first defines a so-called *fitness function* $f : L \to R$ and a set T of transformations, where each transformation $t \in T$ is a mapping $t : L^* \to L$, i.e. a transformation produces an element $t(p_1, \ldots, p_r) \in L$ from elements $p_1, \ldots, p_r \in L$.

The search then proceeds in iterations usually called *generations*. In each generation i a subset P_i of L of a certain cardinality s is generated. Such a subset is called a *population* and its elements are called *individuals*. Typically, the first population is generated in a random fashion and elements in population P_i are generated by applying transformations $t \in T$ to elements $p_1, \ldots, p_r \in P_{i-1}$ in a random fashion. In this step individuals $p_j \in P_{i-1}$ with high fitness are chosen with higher probability than individuals with low fitness.

If L is a set of programs, then typical transformations are

1. *Mutation*: the syntax tree of a single program in P_{i-1} is changed at a small number of places in a random fashion.
2. *Crossover*: two subtrees are swapped between two syntax trees of programs in P_{i-1}.
3. *Hierarchy*: a small new syntax tree is generated in a random fashion. Some leaves of this tree are replaced by syntax trees of programs in P_{i-1}.

For populations $P \subset L$ we denote by $T(P)$ the set of all elements of L which can be generated with a single transformation from elements in P. In order to

motivate the use of genetic programming we show that several observations from psychology can be interpreted in a rather natural way with the help of genetic algorithms.

2.2 Relations to Piaget's Work

Around the year 1900 the Swiss psychologist Piaget published his famous work [Pia91] on child development. Piaget claimed that children acquire new skills in a succession of *stages* and that a new stage can only be entered when the skills of the previous stage have been mastered. A typical example is that movements learned in stage i are repetitively executed in stage $i + 1$. Critics objected that the separation between stages was not clean and that children occasionally *do* learn skills from a later stage before mastering all skills of the previous stage.

Around 1900 computers and programming languages did not exist. With these concepts at hand it is tempting to rephrase Piaget's claim as follows: a *stage* is a set of programs and $stage_{i+1} \subset stage_i \cup T(stage_i)$ for all i and an appropriate set T of transformations. Learning the repetitive execution of a movement learned in stage i then corresponds to the construction of a loop around a program from stage i via the hierarchy transformation.

Let S be a set of programs which correspond to the skills observed by Piaget. For programs $p, p' \in S$ we define $p < p'$ iff p' is generated by a transformation which uses p. Then the transitive closure $<^*$ of relation $<$ is a *partial* order on S which models in a very natural way the order in which *certain* skills have to be acquired.

2.3 Building Theories

In the spirit of [Pau94] we view theories as programs q which reproduce data D from the real world. A theory is nontrivial if it is 'simpler' than the data it describes. A good example are the so-called intelligence tests like

$$D = 2, 3, 4, 6, 8, 12, 14, 18, 20, 24, \ldots$$

The goal is to continue the series. This requires finding as 'simple' as possible a program q, which reproduces the given data.

We consider two candidate solutions for the above example, namely

$$2, 3, 4, 6, 8, 12, 14, 18, 20, 24, 1, 1, 1, \ldots$$

and

$$2, 3, 4, 6, 8, 12, 14, 18, 20, 24, 30, 32, 38 \ldots$$

In all common programming languages L a program q which simply prints the above data D one by one and then outputs 1's is considerably shorter than a program which enumerates the prime numbers + 1. Nevertheless we prefer the second solution. This illustrates two things:

1. The 'Simplicity' of solution q is *not* simply measured by the length $|q|$ of program q alone.

2. There is apparently a set $P \subset L$ of programs - corresponding to mathematical concepts we are already familiar with - that we are allowed to use free of charge. This subset P defines the current inductive bias. Candidate solutions are apparently sought in the set $T(P)$ for some set T of transformations. With hints like: 'Think of prime numbers !' the bias P can be narrowed. Because of the combinatorics involved this in general dramatically narrows $T(P)$ and accordingly solutions are found much faster.

Let L_T be a programming language for specifying programs obtained by transformations in T, i.e. programs in L_T take as input a set P of programs and produce as output a program of the form $t(p_1, \ldots, p_r)$ with $t \in T$ and $p_j \in P$ for all j. For programs $t \in L_T$ and sets $P \subset L$ we denote by $t(P)$ the output of program t started with input P. This output is an element of $T(P)$. Then the building of theories with inductive bias P can be formulated as an optimization problem: Find a transformation $t \in L_T$ such that program $t(P)$ reproduces the given data D. Minimize $|t|$!

This formal definition captures two *subjective* aspects of 'simplicity': what we consider simple depends on what we know (P) and how we generate new concepts from old ones (L_T).

2.4 Occam's Razor

The condition 'Minimize $|t|$!' above is one possible formalization of Occam's Razor.[1] Without a formal model of how new theories are found there is no hope to prove or disprove one or another version of a principle like Occam's Razor. But if we assume that theories are formed by searching a set of the form $T(P)$, then we can indeed derive the above form of Occam's Razor from simpler principles:

If programming languages L and L_T are sufficiently rich, then there is no universally good strategy to search the set $T(P)$ for *any* program that reproduces data D. In this case the fastest strategy is to try as many programs as possible as quickly as possible. If the search is sequential, then short programs should be enumerated before long ones. If the search is performed by parallel units which communicate only after one of them has found a solution, then programs should be tried randomly, but short programs should be tried with a higher probability than long programs. In any case *short solutions tend to be found first.* If we have found a long solution and then are confronted with a short solution we tend to ask:'Why didn't I think of this first ?. The criterion 'Prefer t over t' if t is shorter than t'' now immediately follows from the simpler principle: 'Prefer the solution which you *normally* would have found first' or equivalently: 'Prefer the solution which would be found first

[1] It ignores the run time of programs, but that is easily fixed: restrict $T(P)$ to programs with short run time.

most often, if several persons with the same education simultaneously tried to find a solution'.

3 Schemas

In the spirit of microcomputer architecture we view our learning systems as a bus, which is connected to the outside world via I/O-ports for sensor input and effector output. Internally there is read-only memory containing the initial 'native' program, one or more CPU's and main memory in which an *internal data structure* is built. At each time this data structure is the system's model of the world. The system works in two modes: in *passive* mode the system makes theories about the sensor input using genetic programming. In *active* mode the system performs actions and models the effect of its action using a kind of schema mechanism in the spirit of [Dre91]. A third *introspective* mode has not yet been implemented. In this mode the system should make theories about the internal data structure. This permits planning, and it is at least not completely implausible that the system at some point generates a model of itself.

According to [Dre91] a *schema* is a triple $s = pre/action/post$, where

- pre (precondition) and $post$ (postcondition) are programs which depend on sensor input and compute predicates and
- $action$ is an elementary or composite program which activates effectors.

The semantics of a schema is: If pre holds and $action$ is executed, then possibly $post$ holds; iff the postcondition $post$ holds the activation of the schema is called successful. The reliability of schema s is defined as

$$rel(s) = \#\text{successful activations of } s/\#\text{activations of } s,$$

and the applicability of schema s is defined as

$$app(s) = \#\text{activations of } s/\text{lifetime of } s$$

where lifetime of s is the number of steps[2] the system has made since the schema was generated; the $\#$-sign should be read as 'number of'.

4 Statistical Implication

One of the key operations that can be performed with schemas is *chaining*. If the postcondition $post_1$ of schema s_1 implies the precondition pre_2 of a second schema s_2 then we can chain the two schemas and obtain a schema

$$pre_1/action_1/post_1 \rightarrow pre_2/action_2/post_2$$

[2] to be defined in section 5

with the obvious semantics: if pre_1 holds and actions $action_1$ and $action_2$ are performed, then possibly $post_2$ holds. Of course one can build whole graphs of schemas in this way. Such a graph is a spatial model of the world.

Determining when predicate $post_1$ implies predicate pre_2 of course is not a trivial matter. If we allow arbitrary computable predicates, the question is undecidable. Even if we treat sensor inputs as boolean variables and restrict predicates to boolean formulae of these variables the question remains NP-complete. The solution in [Dre91] is to allow only monomials formed with such variables as predicates. In this case if $post_1$ implies pre_2 iff pre_2 is a submonomial of $post_2$. Thus a simple syntactic check decides, if two schemas can be chained. Unfortunately the restriction to monomials turns out to be too severe, and in [Dre91] many technical so-called 'embellishments' are necessary in order to partially fix this problem.

The easy way out is to allow unreliable edges between schemas. Thus we create an edge $s_1 \rightarrow s_2$ if after activation of schema s_1 the precondition of schema s_2 holds.

5 Standpoint

Chaining permits building directed graphs of schemas, which are spatial - but in general not Euclidean - models of the world. In this graph we introduce a pointer to the schema which was executed last. This pointer is called the *standpoint* and models the system's present position in the world.

The system works in steps in which it randomly chooses between

1. Creating knowledge by generating a new schema
2. Creating knowledge by generating a new edge
3. Testing created knowledge by activating a schema, chosen from the successor schemas of the standpoint with fulfilled preconditions

An edge $e = (s_1, s_2)$ is *traversed* iff the standpoint changes from s_1 to s_2 in one step by choosing s_2 for activation. A traversal is successful iff the activation of s_2 is successful. The reliability of an edge e is defined as

$$rel(e) = \#\text{successful traversals through } e / \#\text{traversals through } e$$

An edge $e = (s_1, s_2)$ is applicable iff the standpoint points to s_1 and the precondition of s_2 holds. The applicability of e is defined as

$$app(e) = \#\text{Situations in which } e \text{ is applicable} / \text{lifetime of } e$$

6 Quality and Boredom

Because the created knowledge will contain errors there is the need for a mechanism to delete "bad" knowledge. The quality of a schema s is defined as

$$qual(s) = rel(s) * app(s)$$

and the quality of an edge e is defined as

$$qual(e) = rel(e) * app(e)$$

If the quality of a schema/edge is below a certain threshold called schema/edge-death, then the schema/edge is deleted. The schema-death is defined as

$$\text{schema-death} = c_1/\#\text{schemas in the graph}$$

and edge-death is defined as

$$\text{edge-death} = c_2/\#\text{edges in the graph}$$

where c_1, c_2 are constants $\in]0; 1]$. Furthermore the Quality influences the random selection of the next schema for activation. The probability of traversing $e = (s_1, s_2)$ and activating s_2 is proportional to $qual(e) * qual(s_2)$.

A cycle in the graph with high quality knowledge leads to a "fidget-behavior", i.e. traversing the edges of the cycle and activating its schemas increase the qualities of those edges and schemas which in turn increases the probability of staying in the cycle. To prevent the system from getting stuck in such a cycle we introduce a variable which measures how long ago the last new knowledge was generated. This variable is called *boredom*. If boredom is high the meaning of quality is turned upside down, i.e. higher quality of the edge $e = (s_1, s_2)$ and schema s_2 leads to a lower probability of traversing e and activating s_2.

7 Experiments

So far, we have tested the techniques described above in three kinds of experiments. Some of them are pure simulations, some use a real robot arm and camera.

7.1 Microworlds

The microworlds we have simulated are 2-dimensional $n \times n$-grids with a single block and a robot hand in a black workspace. The block as well as the hand can be at one of the positions (i, j) with $i, j \in \{1, \ldots, n\}$. For each grid position i, j there is a sensor $hand(i, j)$ which is active when the hand is at position i, j as well as a sensor $block(i, j)$ which is active if the block is at position (i, j). Moreover there is a single sensor $blockinhand$. The logical negation of the precondition of a

schema is used as postcondition because the system should find out which actions in which situations produce any changes. There are 4 elementary actions $hand(d)$ with $d \in \{N, S, E, W\}$ for incrementally moving the hand one grid position to the north, south, east or west. Moreover there are 2 elementary actions $grasp$ and $ungrasp$ for grasping and releasing the block if it is at the position of the hand.

If the block is absent from the microworld the system learns hand-eye coordination by discovering schemas of the form

$$hand(i,j)/hand(N)/not(hand(i,j))$$

which are chained together in a grid like fashion. For the case $n = 4$ the resulting graph is shown in figure 1.

In order to simplify the drawing schemas with identical preconditions are enclosed in dotted rectangles, but the system does not use such syntactical information. A edge drawn towards a rectangle symbolizes edges to all enclosed schemas. Furthermore $hand(i,j)/hand(d)/not(hand(i,j))$ is abbreviated $i,j/d$. The system discovers this structure by a kind of random walk over an $n \times n$-grid. Analyzing the expected time for this walk is not easy, because exploration of new actions depends on the variable boredom. Mean run times from 10 runs on an 20 MHz SPARCstation 1 are reported in table 1. If the block is present, then additional schemas of the following flavor are found

$$hand(i,j) \wedge block(i,j) \wedge not(blockinhand)/grasp/not(precondition)$$

An complication arises, because it is a priori not clear that hand motion is independent of the block. Thus schemas of the form

$$hand(i,j) \wedge block(k,l)/hand(d)/not(precondition)$$

are created. These schemas have high reliability. But as the block is moved around their applicability falls below that of the more general schemas which ignore the block and they eventually die. Table 2 shows mean run times for 10 runs for this simulation on a 40 MHz SPARCstation 10/51.

In [Dre91] a similar but somewhat richer microworld was simulated on a CM-2 parallel machine for the case $n = 5$. There the simulation stopped after 1 day because of memory overflow before the entire structure for hand-eye coordination was found, and no schemas for grasping/ungrasping were discovered.

7.2 Interfacing to the Real World

The experiment with hand-eye coordination was repeated with a real robot arm and a camera for the case $n = 5$. Preprocessors for the vision system and the robot control were written in such a way that the real system could interface to the learning mechanism like the sensor inputs and actor outputs of the microworld. Not surprisingly learning hand-eye coordination worked as expected. Run time was now dominated by the speed of the robot arm and went up a factor of approximately 500.

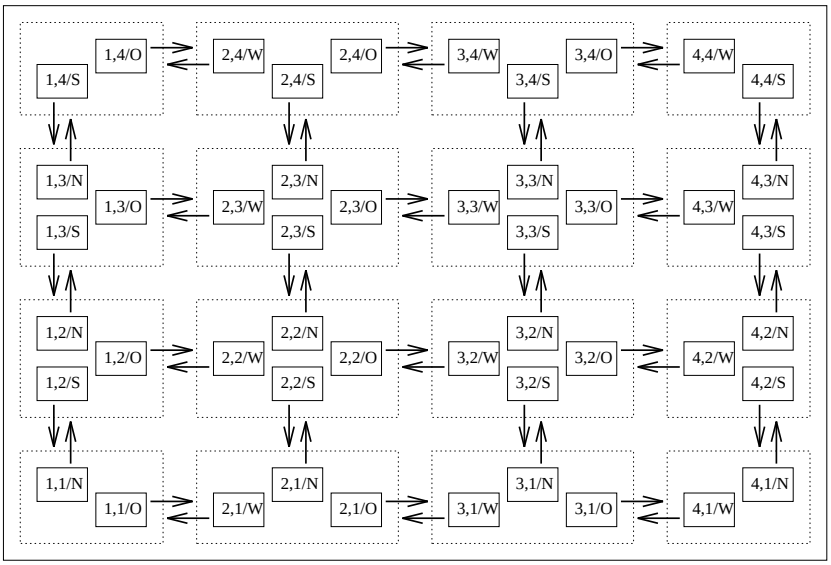

Figure 1: The hand-eye coordination graph for the case $n = 4$

gridsize	2×2	3×3	4×4	5×5	6×6	7×7	8×8
time	0.1	0.7	2.1	7.7	24.5	47.7	1:31.2
gridsize	9×9	10×10	11×11	12×12	13×13	14×14	15×15
time	2:55.1	5:37.1	11:0.5	18:16.5	29:42.4	43:29.2	1:07:46.0

Table 1: Run times (hh:mm:ss) to learn hand-eye coordination

gridsize	2×2	3×3	4×4	5×5	6×6	7×7	8×8	9×9
time	0.2	0.9	6.8	22.6	54.7	1:41.3	6:55.9	13:35.5

Table 2: Run times (mm:ss) to learn hand-eye coordination including grasping

This experiments allowed to exercise the mechanism for getting rid of unreliable edges. For example, holding an object with the shape of the robot hand at position (x, y) under the camera while the real hand moved from position (i, j) created undesired edges like

$$hand(i,j)/hand(d)/not(hand(i,j)) \rightarrow hand(x,y)/hand(d')/not(hand(x,y)).$$

Subsequently these edges proved unreliable and died.

7.3 Discovering Geometric Shapes

The experiments reported so far grossly oversimplify the real world and hardly exercise the mechanism for genetic programming. The next step is to get rid of much of the preprocessing for the vision system and the robot control. For an experiment with a black background, a red triangular robot hand and (non reflecting) blocks the preprocessor of the vision system only classifies each of the 100×100 pixels coming from the camera as having one of 16 colors.

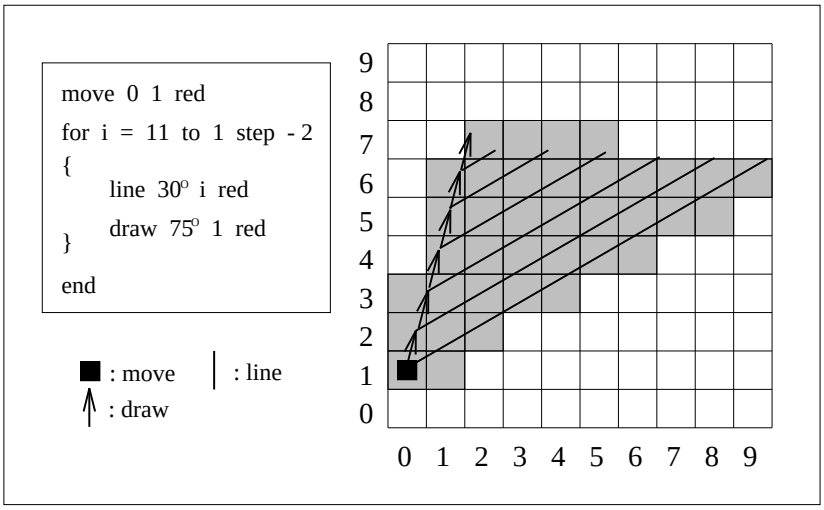

Figure 2: A program producing a red triangle

The system now tries in passive mode (see section 3) to find programs which describe the camera input. This is done with genetic programming and can be viewed as a process of symbol creation. When a program p is used as condition $cond$ in a schema, $cond$ holds if and only if the output of p equals the current camera input.

As the programming language L_p for describing preprocessed pictures we use *turtle graphics*. This is a simple imperative programming language for drawing pictures on an $n \times n$-array of pixels. The specific instruction for drawing lines has

the form $draw\ \alpha\ l\ c$. This command moves an imagined hand holding a pen with color c from its current position on the array for l units in direction of angle α. The instruction $line\ \alpha\ l\ c$ draws the same line as $draw\ \alpha\ l\ c$ and then returns the imagined hand to its original position. We use 72 angles α. The instruction $move\ x\ y\ c$ puts the imagined hand on position (x, y) and assigns color c to this point. Loops can be done with a for command. For programs $p \in L$ we denote by $a(p)$ the picture drawn by program p. Figure 2 shows an example program which produces a red triangle as output.

We use the following function D to measure the difference between two pixel arrays a and a':

$$D(a, a') = \sum_c d^1(a, a', c) + d^2(a', a, c)$$

$$d^1(a, a', c) = \frac{\sum_{a[X]=c} \min\{md(X, Y) | a'[Y] = c\}}{\#_c(a)}$$

$$d^2(a, a', c) = \frac{\max\{0, \#_c(a) - \#_c(a')\}}{\#_c(a)}$$

where $\#_c(a) = \#\{X | a[X] = c\}$ and $a[X]$ denotes the color of pixel array a at position $X = (x_1, x_2)$ and $md(X, Y) = |x_1 - y_1| + |x_2 - y_2|$ is the Manhattan-distance between positions X and Y. D has two important properties. It is computable fast in time linear in the number of pixels. And it has expressive gradients with regard to translation, rotation and expansion of objects. More precisely, given a picture a showing an object and a picture a' showing the same object translated, rotated or expanded. A decrease in the euclidean distance, the rotation angle or expansion factor of the object in a' with regard to its representation in a leads to a decrease of $D(a, a')$.

The task of the genetic programming in the passive mode can now be formulated as the following optimization problem: Given the current camera input as a pixel array a. Find a program p which minimizes $D(a, a(p)) + |p|$. In the following round of the active mode, program p can be used as condition in a newly created schema.

The genetic programming proceeds in the following way: We use as the initial population $P \subset L_p$ randomly generated programs and programs used as conditions in schemas with high quality. Population P_i is generated from population P_{i-1} as described in section 2.1 by using the transformations *reproduction*, *hill-climbing*, *conc* and *split*. *Reproduction* passes an individual from P_{i-1} to P_i unaltered. *Hill-climbing* randomly chooses a constant in a program and performs a hill-climbing-step on it. In doing so the probability of stepping width w is inversely proportional to w. *Conc* concatenates two programs and *split* splits a program in two new ones at a randomly chosen place. We presently do not use crossover.

The passive mode of system discovers various geometric figures [Bir96]. On a 40MHz SPARCstation 10/51 using 100×100 pixel arrays, population size 50, a triangle is found on the average in 23 hours and a quadrangle in 7 hours (both in arbitrary form and orientation). Once the right program (but with the wrong parameters) is in population P, fitting parameters is for example done for a triangle in 5 minutes and for a quadrangle in 4 minutes. These times indicate that once the programs for *one* quadrangle, triangle etc. are found, then recognition of *any* rectangle, triangle etc. will be fast.

Experiments with the complete system, i.e. including active mode, confirm this observation. The hand-eye coordination graph (see figure 1) for the case $n = 5$ was found e.g. in 36 hours when using a graphical simulation of the robot hand. During this run a first schema of the grid was discovered after approximately 27 hours. Thereafter it took on average only 8:30 minutes to find each of the 79 missing schemas.

8 Conclusion and further Work

We described a very general kind of learning systems which simultaneously explore and describe their environment. When they start out, they hardly have any a priori knowledge at all. In these systems the basic units of knowledge are schemas in the sense of [Dre91]. Technically we have modified Drescher's original schema mechanism in three ways:

1. Implication between predicates is purely statistical.
2. The introduction of a standpoint permits to interpret a network of schemas as a spatial model of the environment.
3. Predicates are generated with the help of genetic programming. This is supposed to open the way to process raw input from video cameras.

We have reported the results of computer simulations and of real experiments which successfully exercised different parts of the system. In particular our system discovered several networks of schemas which were not discovered by Drescher's original system, and it got rid of unreliable knowledge.

The obvious next steps are to integrate the whole system, run robot arm and camera with inputs coming directly from the vision system and see what happens. The related experiments and results are described in [**?**].

References

[Bir95] Andreas Birk; Stimulus Response Lernen, ein neues Machine Learning Paradigma; PhD thesis, Universität des Saarlandes, Saarbrücken, 1995

[Bir96] Andreas Birk; Learning Geometric Concepts with an Evolutionary Algorithm; Proc. of The Fifth Annual Conference on Evolutionary Programming; The MIT Press, Cambridge, 1996

[Car90] Jaime Carbonel [Ed.]; Machine Learning, Paradigms and Methods; The MIT Press, Cambridge, 1990

[Dre91] Gary L. Drescher; Made-up minds, A constructivist approach to artificial intelligence; The MIT Press, Cambridge, 1991

[Koz92] John R. Koza; Genetic programming; The MIT Press, Cambridge, 1992

[Nat91] Balas K. Natarajan; Machine Learning, A theoretical approach; Morgan Kaufmann, San Mateo, 1991

[Pau94] Wolfgang J. Paul, R. Solomonoff; Autonomous Theory Building Systems; Annals of Operations Research, Kluwer, 1995

[Pia91] Jean Piaget; Gesammelte Werke; Klett-Cotta, Stuttgart, 1991

[Rit91] Helge Ritter, Klaus Schulten, Thomas Martinetz; Neuronale Netze; Addison Wesley, 1991