A Framework for Behavioural Cloning

Michael Bain

Claude Sammut

Department of Artificial Intelligence,
University of New South Wales,
Sydney Australia 2052
(email: mike@cse.unsw.edu.au, claude@cse.unsw.edu.au)

July 30, 2001

Abstract

This paper describes recent experiments in automatically constructing reactive agents. The method used is behavioural cloning, where the logged data from skilled, human operators are input to an induction program which outputs a control strategy for a complex control task. Initial studies were able to successfully construct such behavioural clones, but suffered from several drawbacks, namely, that the clones were brittle and difficult to understand. Current research is aimed at solving these problems by learning in a framework where there is a separation between an agent's goals and its knowledge of how to achieve them.

1 Introduction

Behavioural cloning has been successfully used to construct control systems in a number of domains (Michie et al, 1990; Sammut et al, 1992; Urbancic & Bratko, 1994). Clones are built by recording the performance of a skilled human operator and then running an induction algorithm over the traces of the behaviour. The most basic form of behavioural cloning results in a set of situation-action rules that map the current state of the process being controlled to a set of actions that achieve some desired goal.

This formulation of the problem has several weaknesses.

- The rule sets generated are very often large and difficult to understand.

- The controllers may not be robust with respect to changes in initial conditions and disturbances in the environment.

In this paper, we describe some attempts to solve these problems. The main theme running through the work described here is that greater structure is added to the problem as compared with the original formulation. In particular, we examine the following techniques:

- decomposing learning into two tasks: learning goals and learning the actions that achieve those goals;
- constructing high-level features;
- providing a mixed-mode of automated learning and interactive knowledge acquisition.

These techniques are illustrated using the domain of learning to fly an aircraft in a flight simulator. The following section describes the original "learning to fly" task and subsequent sections introduce each of the above techniques for structuring the problem.

2 Learning to Fly

Sammut, Hurst, Kedzier and Michie (1992) modified a flight simulation program to log the actions taken by a human subject as he or she flies an aircraft. The log file is used to create the input to an induction program. The quality of the output from the induction program is tested by running the simulator in autopilot mode where the autopilot code is derived from the decision tree formed by induction.

The central control mechanism of the simulator is a loop that interrogates the aircraft controls and updates the state of the simulation according to a set of equations of motion. Before repeating the loop, the instruments in the display are updated.

2.1 Logging Flight Information

The display update was modified so that when the pilot performs a control action by moving the control stick or changing the thrust or flaps settings, the state of the simulation is written to a log file. Three subjects each 'flew' 30 times.

At the start of a flight, the aircraft points North, down the runway. The subject is required to fly a well-defined flight plan that consists of the following manoeuvres:

1. Take off and fly to an altitude of 2,000 feet.

2. Level out and fly to a distance of 32,000 feet from the starting point.

3. Turn right to a compass heading of approximately 330°. The subjects were actually told to head toward a particular point in the scenery that corresponds to that heading.

4. At a North/South distance of 42,000 feet, turn left to head back towards the runway. The scenery contains grid marks on the ground. The starting point for the turn is when the last grid line was reached. This corresponds to about 42,000 feet. The turn is considered complete when the azimuth is between 140° and 180°.

5. Line up on the runway. The aircraft was considered to be lined up when the aircraft's azimuth is less than 5° off the heading of the runway and the twist is less that ±10° from horizontal.

6. Descend to the runway, keeping in line. The subjects were given the hint that they should have an 'aiming point' near the beginning of the runway.

7. Land on the runway.

During a flight, up to 1,000 control actions can be recorded. With three pilots and 30 flights each, the complete data set consists of about 90,000 events. The data recorded in each event are:

on_ground	boolean: is the plane on the ground?
g_limit	boolean: have we exceeded the plane's g limit
wing_stall	boolean: has the plane stalled?
twist	integer: 0 to 360° (in tenths of a degree, see below)
elevation	integer: 0 to 360° (in tenths of a degree, see below)
azimuth	integer: 0 to 360° (in tenths of a degree, see below)
roll_speed	integer: 0 to 360° (in tenths of a degree per second)
elevation_speed	integer: 0 to 360° (in tenths of a degree per second)
azimuth_speed	integer: 0 to 360° (in tenths of a degree per second)

airspeed	integer: (in knots)
climbspeed	integer: (feet per second)
E/W distance	real: E/W distance from centre of runway (in feet)
altitude	real: (in feet)
N/S distance	real: N/S distance from northern end of runway (in feet)
fuel	integer: (in pounds)
rollers	real: ± 4.3
elevator	real: ± 3.0
rudder	real: not used
thrust	integer: 0 to 100%
flaps	integer: $0°$, $10°$ or $20°$

The elevation of the aircraft is the angle of the nose relative to the horizon. The azimuth is the aircraft's compass heading and the twist is the angle of the wings relative to the horizon. The elevator angle is changed by pushing the mouse forward (positive) or back (negative). The rollers are changed by pushing the mouse left (positive) or right (negative). Thrust and flaps are incremented and decremented in fixed steps by keystrokes. The angular effects of the elevator and rollers are cumulative. For example, in straight and level flight, if the stick is pushed left, the aircraft will roll anti-clockwise. The aircraft will continue rolling until the stick is centred. The thrust and flaps settings are absolute.

When an event is recorded, the state of the simulation at the instant that an action is performed could be output. However, there is always a delay in response to a stimulus, so ideally we should output the state of the simulation when the stimulus occurred along with the action that was performed some time later in response to the stimulus. But how do we know what the stimulus was? Unfortunately there is no way of knowing. Human responses to sudden piloting stimuli can vary considerably but they take at least one second. For example, while flying, the pilot usually anticipates where the aircraft will be in the near future and prepares the response before the stimulus occurs.

Each time the simulator passes through its main control loop, the current state of the simulation is stored in a circular buffer. An estimate is made of how many loops are executed each second. When a control action is performed, the action is output, along with the state of the simulation as it was some time before. How much earlier is determined by the size of the buffer.

2.2 Data Analysis

Quinlan's C4.5 (Quinlan, 1993) program was used to generate flight rules from the data. Even though induction programs can save an enormous amount of human effort in analysing data, in real applications it is usually necessary for the user to spend some time preparing the data.

The learning task was simplified by restricting induction to one set of pilot data at a time. Thus, an autopilot has been constructed for each of the three subjects who generated training data. The reason for separating pilot data is that each pilot can fly the same flight plan in different ways. For example, straight and level flight can be maintained by adjusting the throttle. When an airplane's elevation is zero, it can still climb since higher speeds increase lift. Adjusting the throttle to maintain a steady altitude is the preferred way of achieving straight and level flight. However, another way of maintaining constant altitude is to make regular adjustments to the elevators causing the airplane to pitch up or down.

The data from each flight were segmented into the seven stages described previously. In the flight plan described, the pilot must achieve several, successive goals, corresponding to the end of each stage. Each stage requires a different manoeuvre. Having already defined the sub-tasks and told the human subjects what they are, the learning program was given the same advantage.

In each stage four separate decision trees are constructed, one for each of the elevator, rollers, thrust and flaps. A program filters the flight logs generating four input files for the induction program. The attributes of a training example are the flight parameters described earlier. The dependent variable or class value is the attribute describing a control action. Thus, when generating a decision tree for flaps, the flaps column is treated as the class value and the other columns in the data file, including the settings of the elevator, rollers and thrust, are treated as ordinary attributes. Attributes that are not control variables are subject to a delay, as described in the previous section.

C4.5 expects class values to be discrete but the values for elevator, rollers, thrust and flaps are numeric. A preprocessor breaks up the action settings into sub-ranges that can be given discrete labels. Sub-ranges are chosen by analysing the frequency of occurrence of action values. This analysis must be done for each pilot to correctly reflect differing flying styles. There are two disadvantages to this method. One is that if the sub-ranges are poorly

chosen, the rules generated will use controls that are too fine or too coarse. Secondly, C4.5 has no concept of ordered class values, so classes cannot be combined during the construction of the decision tree.

An event is recorded when there is a change in one of the control settings. A change is determined by keeping the previous state of the simulation in a buffer. If any of the control settings are different in the current state, a change is recognised. This mechanism has the unwanted side-effect of recording all the intermediate values when a control setting is changed through a wide range of values. For example, the effects of the elevator and rollers are cumulative. If we want to bank the aircraft to the left, the stick will be pushed left for a short time and then centred, since keeping it left will cause the airplane to roll. Thus, the stick will be centred after most elevator or roller actions. This means that many low elevator and roller values will be recorded as the stick is pushed out and returned to the centre position.

To ensure that records of low elevator and roller values do not swamp the other data, another filter program removes all but the steady points and extreme points in stick movement. Control engineers are familiar with this kind of filtering. In their terms, the graph of a control's values is differentiated and only the values at the zero crossings of the derivative are kept.

2.3 Generating the Autopilot

After processing the data as described above, they can be submitted to C4.5 to be summarised as rules that can be executed in a controller.

Decision tree algorithms are made noise tolerant by introducing *pruning*. If the data contain noise, then many of the branches in a decision tree will be created to classify bad data. The effects of noise can be reduced by removing branches near the leaves of the tree. This can either be done by not growing those branches when there are insufficient data or by cutting back branches when their removal does not decrease classification accuracy.

The flight data are very noisy, so decision trees are generated using conservative setting for pruning and then tested in the simulator. Pruning levels are gradually increased until the rule 'breaks', ie. it is no longer able to control the plane correctly. This procedure results in the smallest, and thus most readable, rule the succeeds in accomplishing the flight goal.

2.4 Linking the Autopilot with the Simulator

To test the induced rules, they are used as the code for a autopilot. A post-processor converts C4.5's decision trees into if-statements in C so that they can be incorporated into the flight simulator easily. Hand-crafted C code determines which stage the flight has reached and decides when to change stages. The appropriate rules for each stage are then selected in a switch statement. Each stage has four, independent if-statements, one for each action.

When the data from the human pilots were recorded, a delay to account for human response time was included. Since the rules were derived from these data, their effects should be delayed by the same amount as was used when the data were recorded. When a rule fires, instead of letting it effect a control setting directly, the rule's output value is stored in a circular buffer. There is one for each of the four controls. The value used for the control setting is one of the previous values in the buffer. A lag constant defines how far to go back into the buffer to get the control setting. The size of the buffer must be set to give a lag that approximates the lag when the data were recorded.

Rules could set control values instantaneously as if, say, the stick were moved with infinite speed from one position to another. Clearly this is unrealistic. When control values are taken from the delay buffer, they enter another circular buffer. The controls are set to the average of the values in the buffer. This ensures that controls change smoothly. The larger the buffer, the more gentle are the control changes.

2.5 Flying on Autopilot

An example of the rules created by cloning is the elevator take-off rule generated from one pilot's data:

```
elevation > 4 :  level_pitch
elevation <= 4 :
|   airspeed <= 0 :  level_pitch
|   airspeed > 0 :  pitch_up_5
```

This states that as thrust is applied and the elevation is level, pull back on the stick until the elevation increases to 4. Because of the delay, the final

elevation usually reaches 11 which is close to the values usually obtained by the pilot. pitch_up_5 indicates a large elevator action, whereas, pitch_up_1 would indicate a gentle elevator action.

A more complex case is that of turning. Stage 4 of the flight requires a large turn to the left. The rules are quite complex. To make them understandable, they have been greatly simplified by over-pruning. They are presented to illustrate an important point, that is that rules can work in tandem although there is no explicit link between them. The following rules are for the rollers and elevator in the left turn.

```
azimuth > 114 :  right_roll_1
azimuth <= 114 :
|   twist <= 8 :  left_roll_4
|   twist > 8 :  no_roll

twist <= 2 :  level_pitch
twist > 2 :
|   twist <= 10 :  pitch_up_1
|   twist > 10 :  pitch_up_2
```

A sharp turn requires coordination between roller and elevator actions. As the aircraft banks to a steep angle, the elevator is pulled back. The rollers rule states that while the compass heading has not yet reached 114, bank left provided that the twist angle does not exceed 8. The elevator rule states that as long as the aircraft has no twist, leave the elevator at level pitch. If the twist exceeds 2 then pull back on the stick. The stick must be pulled back more sharply for a greater twist. Since the rollers cause twist, the elevator rule is invoked to produce a coordinated turn. The profile of a complete flight is shown in Figure 1.

Like Michie, Bain and Hayes-Michie (1990), this study found a "clean-up effect". The flight log of any trainer contains many spurious actions due to human inconsistency and corrections required as a result of inattention. It appears that the effects of these inconsistent examples are pruned away by C4.5, leaving a control rule which flies very smoothly.

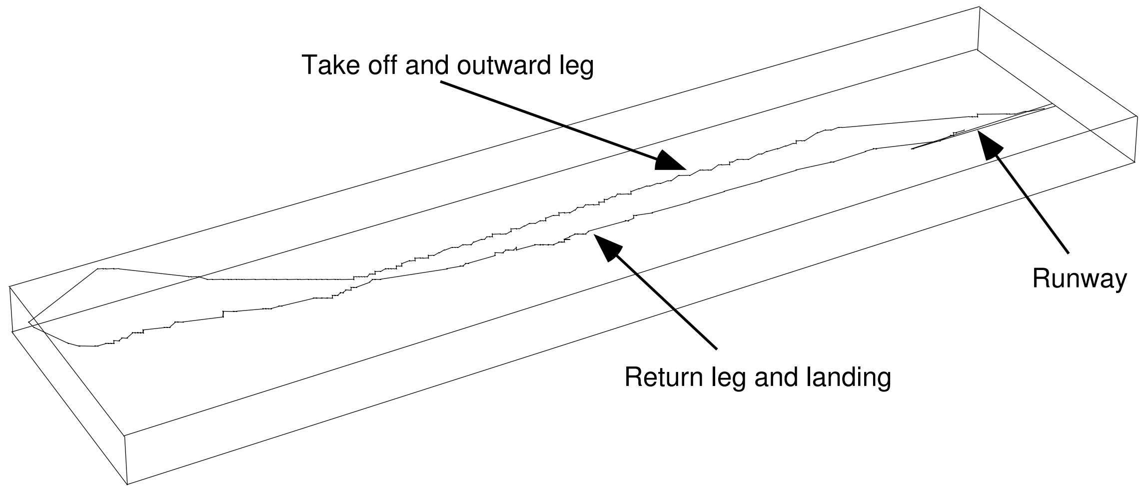


Figure 1: Flight profile.

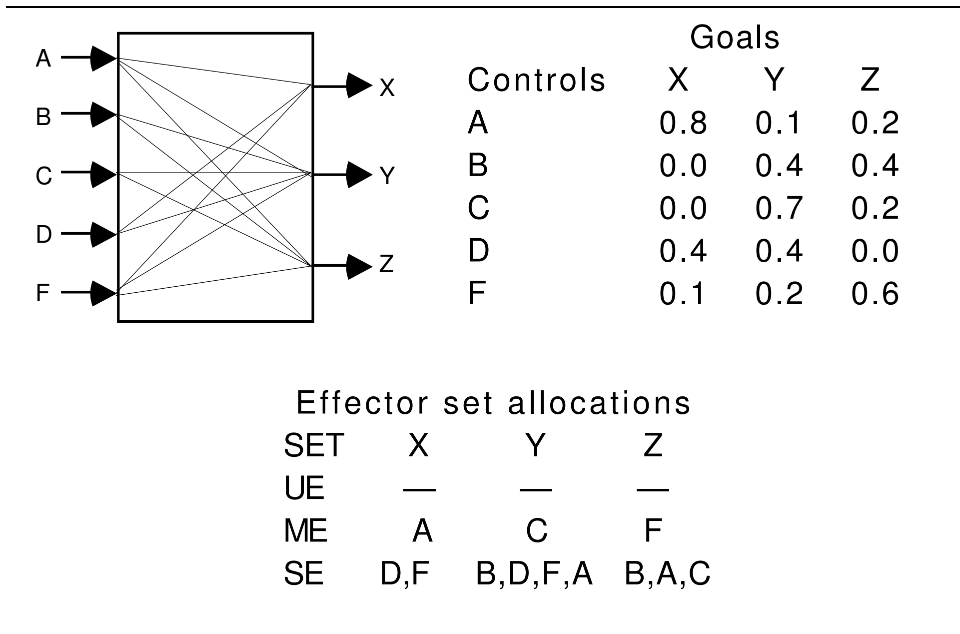
3 Learning to Achieve Goals

One of the interesting features of behavioural cloning is that the method can develop working controllers that have no representation of goals. The rules that are constructed are pure situation-action rules, i.e. they are reactive. However, this feature also appears to result in a lack of robustness. When a situation occurs which is outside of the range of experience represented in the training data, the clone can fail entirely. To some extent, a clone can be made more robust by training in the presence of noise. However, because the clone does not have a representation of how control action can achieve a particular goal, it cannot choose actions in a flexible manner in totally new situations.

3.1 CHURPS

CHURPS (or Compressed Heuristic Universal Reaction Planners) were developed by Stirling (1995) as a method for capturing human control knowledge. Particular emphasis was placed on building robust controllers that can even tolerate actuator failures.

Where behavioural cloning attempts to avoid questioning an expert on their behaviour, Stirling's approach is to obtain from the expert a starting point from which a controller can be generated automatically. The expert



Controls	Goals X	Goals Y	Goals Z
A	0.8	0.1	0.2
B	0.0	0.4	0.4
C	0.0	0.7	0.2
D	0.4	0.4	0.0
F	0.1	0.2	0.6

Effector set allocations

SET	X	Y	Z
UE	—	—	—
ME	A	C	F
SE	D,F	B,D,F,A	B,A,C

Figure 2: A CHURPs model.
(a) An example of a process (b) perceived influences between control inputs and output goals (c) agent's effector view of system.

is asked to supply "influence factors". These are numbers in the range 0 to 1 which indicate how directly a control input affects an output goal. This is illustrated in Figure 4. Here, control action, A, has an influence of 0.8 on goal variable, X. This means that A is the main effector that influences that value of the measure variable, X. Action A also has lesser effects on variables Y and Z. A is also classed as the main effector for goal variable X. From the influence matrix, control actions are grouped into three sets for each goal variable:

Unique Effector (UE) is the only effector which has any influence on a goal variable.

Maximal Effector (ME) has the greatest influence over a particular goal variable. However, other effectors may have secondary influence over that goal variable.

Secondary Effectors (SE) are all the effectors for a goal variable, except the main effector.

The UE, ME and SE sets are used by Stirling's Control Plan generator (CPG) algorithm to generate operational control plans. The algorithm assigns appropriate effectors to control various output goals in order of importance. Informally, the CPG algorithm is:

```
Create an agenda of goals which consist of output variables
whose values deviate from a set point.
The agenda may be ordered by the importance of the goal variable.
  while the agenda is not empty
    select the next goal
    if deviation is small then
      attempt to assign an effector in the order, UE, SE, ME.
    if deviation is large then
      attempt to assign an effector in the order, UE, ME, SE.
    examine influencee's of the effector that was invoked and
      add them to the agenda.
    remove selected goal.
```

The selection of an effector is qualified by the following conditions:

- A controller which is a UE of one goal should not be used as an ME or SE for another goal.
- When choosing an SE, it should be one that has the least side-effects on other goal variables.

This procedure tells us which control actions can be used to effect the desired change in the goal variables. However, the actions may be executed in a variety of ways. Stirling gives the following example. Suppose variables X, Y and Z in Figure 4, are near their desired values. We now wish to double the value of Y while maintaining X and Z at their current levels. Following the CPG algorithm:

1. Y initially appears as the only goal on the agenda.
2. Y had no unique effector and since the required deviation is large, we try to apply an ME, namely, C.
3. Since C also affects Z, Z is appended to the agenda.
4. Since Y is the current goal and an effector has successfully been assigned to it, Y is removed from the agenda.
5. Z becomes the current goal. Let us assume that the deviation in Z is small.
6. We attempt to assign as SE to control Z. B is selected since C is already assigned to control Y. A could have been selected, but it would have a side effect on variable X, causing a further expansion in the agenda.
7. The agenda is now empty and terminates with the assignments {Y/C, Z/B}, which can be read as "control goal Y to its desired state via effector C and control goal Z to its desired state via effector B".

This plan can be executed sequentially, by first using C to bring Y to its desired value and then using C to bring Z to its desired value. A loop would sample the process at regular intervals terminating each phase when the desired value is reached. Alternatively, both actions could be executed in parallel. The first strategy corresponds to one that might be followed by a novice, whereas experts tend to combine well practised behaviours since they do not have to think about them.

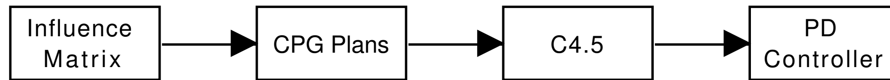


Figure 3: CHURPS Architecture.

As a model of human sub-cognitive skill, the CPG method does not capture the notion that pre-planning is not normally carried out. That is, a skilled operator would not think through the various influences of controls on outputs, but would act on the basis of experience. This is usually faster than first trying to produce a plan and then executing it. To try to simulate this kind of expert behaviour, Stirling used the CPG algorithm as a plan generator which exhaustively generated all combinations of actions for different possible situations. This large database of plans was then compressed by applying machine learning to produce a set of heuristics for controlling the process. The architecture of this system is shown in Figure 3.

To create the input to the learning system (Quinlan's C4.5) each goal variable was considered to have either a zero, small or large deviation from its desired value. All combinations of these deviations were used as initial conditions for the CPG algorithm. In addition, Stirling considered the possibility that one or more control action could fail. Thus plans were also produced, for all of the combinations of deviations and all combinations of effector failures.

Stirling devised a "goal centred" control strategy in which learning was used to identify the effectors that are required to control particular goal variables. Thus if there is a deviation in goal variable Y, a decision tree is built to identify the most appropriate control action, including circumstances in which some control actions may not be available due to failure. An example of a tree for goal variable X, is shown below:

```
if (control A is active)
   if (deviation of X is non-zero)
      use control A
   else
      if (control D is inactive)
         use control A
      else
         if (control F is active)
            use control D
```

```
        else
            use control A
else
    if (control D is active)
        use control D
    else
        use control F
```

Once the control action has been selected, a conventional proportional controller is used to actually attain the desired value.

The CHURPS method has been successfully used to control a simulated Sendzimir cold rolling mill in a steel plant. It has also been used to control the same aircraft simulation used by Sammut *et al.* Like that work, the flight was broken into seven stages. However, one major difference is that CHURPS required the goals of each stage to be much more carefully specified than in behavioural cloning. For example, the original specification of stage 4, the left turn was:

> At a North/South distance of 42,000 feet, turn left to head back to the runway. The turn is considered complete when the compass heading is between 140 and 180

In CHURPS this is translated to

> At a North/South distance of 42,000 feet, establish a roll of 25 ± 2 and maintain pitch at 3 ± 5, airspeed at 100 knots ± 40 knots and climb speed at 1 ft/sec ± 5 ft/sec.
>
> When the plane's compass heading is between 140 and 180, return the roll to 0 ± 2 and maintain all other variables at the same values.

Recalling that the influence matrix was constructed by hand, CHURPS requires much more help from the human expert than behavioural cloning. However, so far, CHURPS have produced smoother and more robust controllers. The question arises, can some combination of behavioural cloning and the CHURPS method used to produce robust controllers requiring minimal advice from the expert?

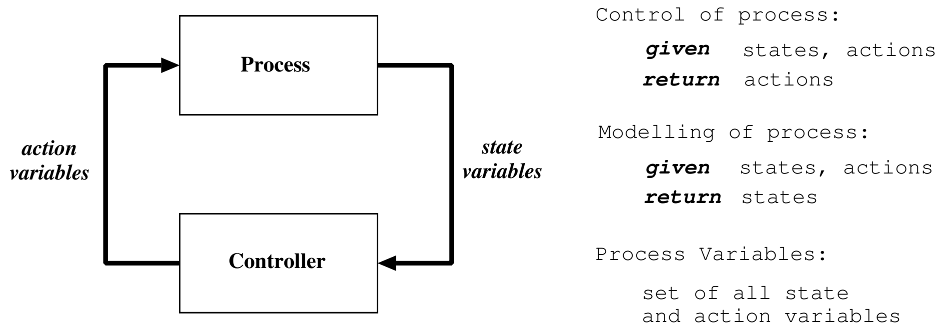


Figure 4: Schematic diagram of a simplified control system.

4 Learning Effects and Goals with Behavioural Cloning

In this section we discuss work on extending the framework of behavioural cloning. A simplified scheme for a control system is assumed, such as that shown in Figure 4. In this scheme the process is a black-box and the controller is an autonomous agent whose memory contains a set of control rules and a buffer of process variables. The original formulation of the behavioural cloning technique requires learning the rules of the form:

$$\textit{action-variable} \leftarrow \textit{process-variables}$$

The antecedent is a subset of the set of all process variables, and may include state and action variables. Therefore a set of such rules is an example of process control as depicted in Figure 4. Usually standard machine learning algorithms for classification are used to learn a behavioural clone. The induced rule-set or theory partitions the space defined by the set of all process variables, classifying each region of this space in terms of the action typically applied by a skilled operator. Reactive control can then be implemented by installing the rules in a controller as in Figure 4 to output actions given process states and actions.

However, for complex control tasks the use of "classical" behavioural cloning presents problems (Arentz, 1995; Urbančič and Bratko, 1994). What is perhaps worse, the successful execution by a clone of even a relatively simple control task can result in behaviour which appears "mindless" (Michie, 1995).

4.1 GRAIL

To address these shortcomings we have re-formulated behavioural cloning in a method called GRAIL, which stands for **G**oal-directed **R**eactive **A**bduction from **I**nductive **L**earning. A GRAIL controller comprises an **effects level** and a **goals level**. Both levels are based on theories built by inductive learning from the traces of skilled operators. In this sense the technique of behavioural cloning is continued in the new method. However the method extends behavioural cloning as follows. Rule sets can be hierarchical, or structured. Also, we allow for the possibility of adding user-supplied rules to the theories. This is mainly intended for adding high-level rules about the control task at the goals level. Additionally we expect that user-supplied or machine-invented predicates ("controller variables") may be useful in extending the vocabulary for describing process variables and controller states. These extensions could apply at both the effects level and the goals level. So far they have not been used in our experiments, since we have concentrated on the learning of theories for each of the effects and goals levels. The theory for the combined effects and goals levels will be referred to as the "task theory". The method is summarised in Figure 5 as an algorithm sketch.

4.1.1 Inductive Learning of task theories

The target theories to be learned at each level are slightly different from those of classical behavioural cloning. At the effects level we have rules of the form:

$$\textit{state-variable} \leftarrow \textit{process-variables}$$

An effects theory can be thought of as approximating *the operator's model of the effects on the process of applying certain control-actions.* As such it is a form of operator-centred process model as depicted in Figure 4.

The goals level is intended to enable the incorporation of rules referring not only to states of the process but also states of the controller. In particular, reference to the goals of the controller is allowed. To this end we suppose a set of controller variables distinct from the process variables of Figure 4. The combined set of process variables and controller variables will be referred to as "task variables." Therefore at the goals level we have rules of the form:

$$\textit{task-variable} \leftarrow \textit{task-variables}$$

These rules may include variables from plans or other background knowledge, or they may contain only process variables. A goals theory can be thought of as approximating *the operator's model of the goals directing their control of the process at any given time.*

As for behavioural cloning, the inductive learning step of GRAIL is done offline from recorded traces of the execution of control tasks by skilled operators. The induced rules are in the form of definite clauses, and the task theory can therefore be understood as a logic program. Our work so far has dealt only with rules containing propositional variables, although we discuss below ways in which first-order learning methods may be used to improve our approach.

4.1.2 Goal-directed Reactive Abduction

To implement control we take advantage of the fact that the theories for both effects and goals are logic programs. The task theory is structured so that actions which can be performed by the controller are included in the bodies of effects rules as "abducible goals", i.e. goals (in the logic programming sense) which can be "made" true (by executing the associated control action). The rules in the remainder of the task theory reduce higher-level goals to lower-level goals using Prolog-style execution.

The controller is presumed to be an autonomous agent linked to a black-box process which is to be controlled. The controller possesses a knowledge base containing the task theory. This knowledge base also contains facts about: the current state of the process (state variables); the current action settings (action variables); possibly other sensory or perceptual information; and a history of previously known facts. These facts are updated at predefined regular time intervals. The time between successive intervals is referred to as the sample period. The sample period is assumed to be sufficient for execution of the task theory with respect to the updated knowledge base, as follows.

Within each sample period a top-level Horn goal $\leftarrow G$ which represents the controller's current task is invoked on the updated knowledge base. Using Prolog-style execution this top-level goal is reduced to a set of low-level goals. In our experiments to date the goals theory is constructed so as to always reduce to a set of $goal_variable = value$ expressions, where each *goal_variable* is one of a predefined set based on a subset of the process variables. These are the low-level goals of Figure 5.

The GRAIL method:

Offline stage: (Inductive Learning)
  From behavioural traces learn theories for:
  - *goals level;*
  - *effects level.*

Online execution: (Goal-directed Reactive Abduction)
  During each sample period:
  - *update values of state variables in knowledge base;*
  - *update top-level goal, then use it to derive low-level goals by backward-chaining on task theory;*
  - *<u>for</u> each of the low-level goals <u>do</u>*
    - *<u>if</u> low-level goal is:*
      an action expression <u>then</u>
        action = goal value;
      an effects expression <u>then</u>
        actions = select-rule(effects expression);
      an indirect-effects expression <u>then</u>
        derive an effects expression;
        actions = select-rule(effects expression);
  - *controller applies actions to process;*
  - *update knowledge-base to record actions applied.*

<u>select-rule</u>(effects = val)
  Let R be the set of effects rules in knowledge base.
  Find $r_i \in R$ such that head(r_i) is "effects = val_i", each condition "state-variable = val_s" in body(r_i) is satisfied in knowledge base and $|val - val_i|$ is minimised for all such r_i.
  If there is more than one such r_i, pick the one with highest coverage on training data.
  <u>return</u> set of conditions "action-variable = val_a" from body(r_i).

Figure 5: GRAIL: a method for behavioural cloning.

The low-level goals are the "set points" for the controller. If the low-level goal is an action expression, i.e. *goal_variable* relates to an action variable, then an assignment of *value* to the corresponding action variable is made. For example, the statement `goal_throttle = 100` leads to the assignment `throttle = 100`. Otherwise, the low-level goal is an effects expression or an indirect-effects expression. This is explained as follows.

Before learning any rules, a representation must be chosen. Process variables are usually pre-determined. However, we must select which of these variables to include in the effects theory. Usually, this requires some knowledge of the domain, or is subject to a degree of trial-and-error. A state-variable selected to be the target-attribute for learning will appear in the heads of a set of effects rules and is referred to as an *effects_variable*. An effects expression is of the form *effects_variable* = *value*. An indirect-effects expression involves a state variable other than an *effects_variable*, from which an effects expression can be derived using a user-supplied pre-defined procedure.

For example, in our experiments in the flight domain it was found convenient to use the low-level goal *goal_elevation*, but to learn effects rules for *elevation_speed*. By taking the difference

$$goal_elevation_speed = goal_elevation - elevation$$

we derive an effects expression from the indirect-effects expression in terms of *goal_elevation*. The effects expression in terms of *goal_elevation_speed* is then used to select an effects rule (see Figure 5).

In the remainder of this section we give examples of learning effects and goals in the flight domain, and discuss the relations between our method and other approaches.

4.2 Learning effects

At this level we require a rule-based model of the effects on certain state variables of control actions. As for the goals level of our controller, the effects rules are inductively learned from trace examples. In the case of flight the system variables can be subdivided into a number of distinct types. For example, the orientation of the aircraft can be described in terms of pitch, roll and yaw. The corresponding controls are elevators, ailerons and rudder. In our simulator the position is slightly simplified by disabling rudder (on advice that its operation is incorrectly simulated). Consequently changes

in yaw, or heading, are treated as side-effects of changes in roll and pitch (twist and roll).

In a simplified formulation of an effects model we have a set of rules of the form

```
Effects_variable = Effects_value ←
            Action_variable = Action_value
```

The action variables are abducibles, in the following sense. Given a desired effect and a rule in the model whose head matches the effect, the control variable is assigned a value which will "cause" the required effect. The rule body could also contain extra literals imposing conditions under which the action will cause the effect.

As an example, take a simple theory for elevation speed defined in terms of elevators. This was induced from instances from the trace in Figure 6. Note that since the elevators are a *rate* controller, the effects variable chosen is elevation speed. This can be linked to the more natural goal of elevation by differencing between target and actual values as described above.

```
Elevation_speed = 3 ← Elevators = -0.28
Elevation_speed = 1 ← Elevators = -0.19
Elevation_speed = 0 ← Elevators = 0.0
Elevation_speed = -1 ← Elevators = 0.9
```

Data was preprocessed using AWK to select variables. Learning was carried out using C4.5. Decision trees were then converted into rules for particular abducibles by AWK scripts, as our rules have a more complex syntax than that generated by C4.5rules.

A similar effects rule was found for the relation between rollers and roll speed. The picture is more complicated when it comes to other effects in the domain, such as airspeed. Airspeed is mainly influenced by throttle, although this is conditional on elevation and other system variables. Additionally, the time delay in the effect of throttle changes on airspeed seems to be greater than the delay in other effects. This is the subject of our current work on learning effects.

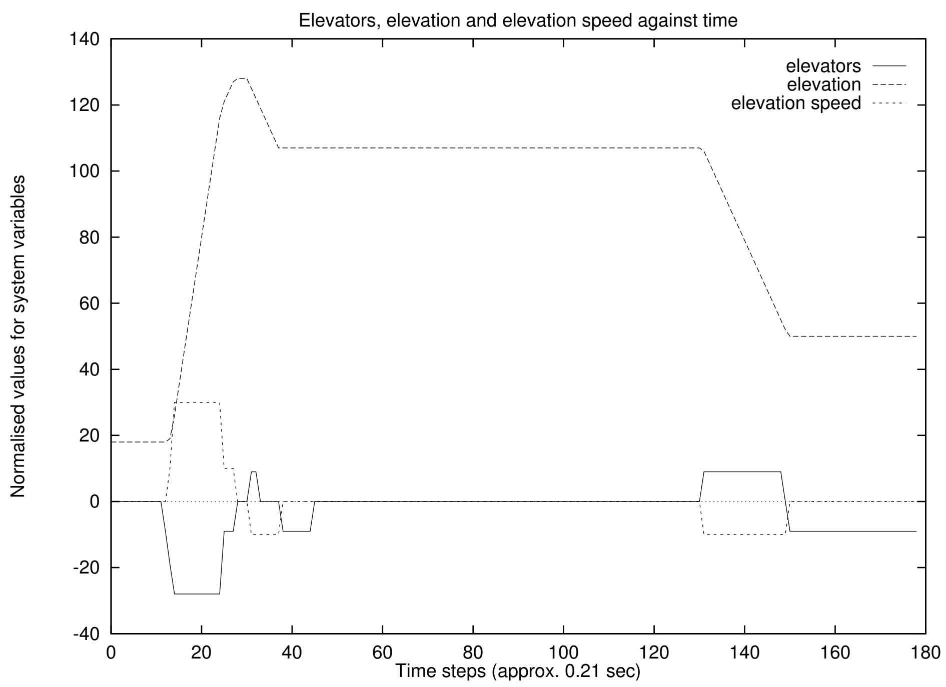


Figure 6: Comparing the effects of elevators on elevation and elevation speed.

4.3 Learning goals

The problem of learning goals can be seen in terms of conjecturing which variables must be attended to and what values must be assigned to those variables in order to achieve the desired outcomes for system control. Clearly this is a difficult task. However, unless goals can be specified in sufficient detail possession of a robust and accurate effects model will not be enough to implement the complex behaviours required. In the flight domain we have begun to learn goals rules which determine system variables in terms of external environment variables. An example theory of this type is given below.

```
if ( Distance > -4007.66 ) Goal_elevation = 0;
else if ( Height > 1998.75 ) Goal_elevation = 20;
else if ( Height > 1918.65 ) Goal_elevation = 40;
else if ( Height > 67.61 ) Goal_elevation = 100;
else if ( Distance <= -4153.4 ) Goal_elevation = 40;
else Goal_elevation = 20;
```

This work is in a preliminary stage, but we hope to improve the method of learning goals in a number of ways. For example, in the example above elevation is set relative to distance from and height above the runway. While this may be adequate for certain manoeuvres, in general it is not a sufficiently powerful representation, since it lacks many of the features a human pilot might use to set goals. Below we discuss how this could be extended where necessary to include high-level features based on background knowledge and relational information pertaining to visual perception.

4.4 Control with effects and goals

Currently the GRAIL approach to behavioural cloning is in development. However, we have evidence from initial investigations that it allows for machine-learned rule-based controllers to be “cloned” from trace examples, and that the induced theories are more compact on a lines-of-code measure than those obtained by a previous behavioural cloning method.

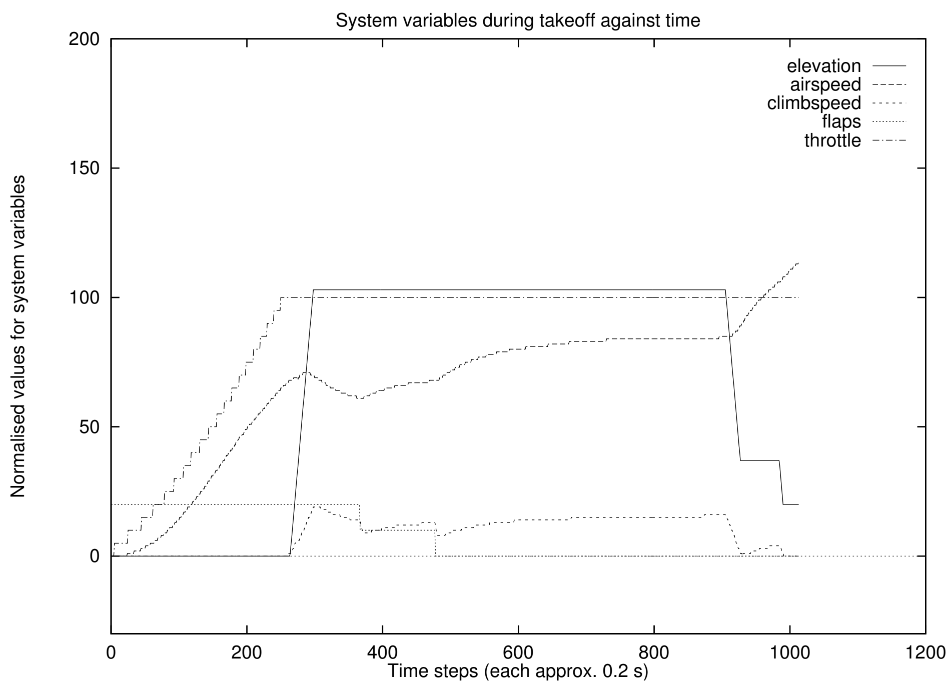


Figure 7: Comparing the system variables during takeoff.

Evaluation (Takeoff)	Cloning method	
	Traditional	GRAIL
Theory size	221	65
Examples	1804	1014
Traces	30	1

The figures in the comparison of traditional with GRAIL behavioural cloning are for the first stage only of the standard flight plan, i.e take off and climb to an altitude of 2000 feet then level out . Theory sizes are measured in lines of C program code [1]. Note that only one trace is required for the GRAIL method compared with thirty for the traditional method, although the total number of examples used are of the same order of magnitude. This is due to the different sampling schemes employed. In (Sammut et al., 1992) an example was recorded only when the pilot changed the setting of one of the four control variables. In contrast, due the requirement of building an effects model for the GRAIL method, a fixed rate sampling scheme is employed. In the current work a sample was recorded approximately once every 0.2 seconds.

We have also used GRAIL to learn simple but general manoeuvres such as climbs and turns. Currently we are working use GRAIL to complete the most difficult flight plan stage accomplished by the traditional method, namely the approach to landing. GRAIL has not yet matched the performance of the earlier method by landing, but its approach to the runway is reasonable and, as seen from the table below, the theory sizes are smaller.

Evaluation (Landing)	Cloning method	
	Traditional	GRAIL
Theory size	8542	680
Examples	8428	3072
Traces	30	5

4.5 Further Work

Whilst the flight of the GRAIL clones has not yet fully matched that of the earlier "best clone" built by the traditional method, it does already have the advantages described above in terms of reduced sizes of rule sets.

[1] This tends to overestimate the complexity of the theories compared with their C4.5 representation but by a factor of less than two.

We have some reason to suppose it may also prove more robust to variations in initial conditions during testing when compared to the traditional method, although this needs to be substantiated. However, we believe it is the possibility of extending GRAIL to use structured theories and background knowledge via first-order learning that is most likely to further improve the performance of behavioural cloning.

In other related work Benson (1996) has adapted the framework of teleoreactive programs for agent control proposed by Nilsson (1994) to the flight domain. This method has some very interesting aspects for agent control, such as the modelling of durative actions and the use of a circuit semantics. The machine learning component of Benson's thesis addresses learning action models for use in the control of a teleo-reactive agent. The agent planning system utilises a formalism for operators called TOPs (for teleo operators). The TOPs framework is closely related to effects rules in GRAIL, but includes the ability to represent the effects of durative actions. TOPs also allow for representation of side-effects in terms of state changes due to the application of actions. However, the application to the flight domain only covered a subset of the stages of the standard flight plan used in our behavioural cloning experiments.

Interestingly, Benson (1996) noted that one difficulty with the application of his learning method to the flight domain was the lack of any temporal reasoning ability in the teleo-reactive formalism. Kowalski (1995) has proposed a framework for combining reactive and rational agency in work which uses abduction to realise agent actions. The method is similar to the goal-directed reactive abduction approach of GRAIL. However, the meta-logical approach of Kowalski provides a very general and powerful framework for planning and reacting which uses an explicit representation for time or resources. Additionally, knowledge assimilation is incorporated via the mechanism of integrity constraints. Aspects of this logic programming framework for agency could provide a basis for methods of first-order learning to be used in behavioural cloning.

5 Constructing High-level Features

Decomposing learning into two stages is one way of structuring the problem domain so that more effective behavioural clones can be built. Another, complimentary approach is to construct high-level features that improve

Table 1: Background Predicates

pos(P, T)	position, P, of aircraft at time, T.
before(T1, T2)	time, T1, is before time, T2.
regression(ListY, ListX, M, C)	least-square linear regression, which tries to find for the list of X and Y values.
linear(X, Y, M, C)	linear(X, Y, M, C) :- Y is M*X+C.
circle(P1, P2, P3, X, Y, R)	fits a circle to three points, specifying the centre (X, Y) and radius, R
$\leq$, $\geq$, abs	Prolog built-in predicates

the expressiveness of the language used to describe the control strategies.

In the original "learning to fly" experiments, only the raw data from the simulator were presented to the learning algorithm. While these data are complete in the sense that they contain all the information necessary to describe the state of the system, they are not necessarily presented in the most convenient form. For example when a pilot is executing a constant rate turn, it makes sense to talk about trajectories as arcs of a circle. Induction algorithms, such as C4.5, can deal with numeric attributes to the extent that they can introduce in equalities, but they are not able to recognise trajectories as arcs or recognise any other kind of mathematical property of the data.

Srinivasan and Camacho (1998) have shown how such trajectories can be recognised by making use of background knowledge with Progol (Muggleton, 1995). The program was applied to the problem of learning to predict the roll angle of an aircraft during a constant rate turn at a fixed altitude. To do this effectively, the target concept must be able to recognise the trajectory as an arc of a circle. The predicates shown in Table 1 are included in the background knowledge[2].

The pos predicate is the input to the learner since it explicitly describes the trajectory of the aircraft as a sequence of points in space. These points are derived from flight logs. The before predicate imposes an ordering on the points in the trajectory. The mode declarations in Srinivasan's version of Progol are not typical of the declarative bias found in other ILP systems.

[2]In practice, it is necessary to include error terms since the regression equation is unlikely to fit new data exactly. However, we omit these here for the sake of clarity

Srinivasan's modes permit the user to specify that some arguments should be lists of values collected over the entire data set. Thus, the mode declaration for regression specifies that the first two arguments are lists which described the sequence of pairs of coordinates for the aircraft during the turn. That is, the coordinates from all the examples in the data set are collected. The mode declaration causes Progol to generate these lists and invokes regression which performs a least-square regression to find the coefficients of the linear equation which relates roll angle and radius. Regression must be accompanied by another background predicate, linear, which implements the calculation of the formula. The theory produced is:

```
roll_angle(Radius, Angle) :-
    pos(P1, T1), pos(P2, T2), pos(P3, T3),
    before(T1, T2), before (T2, T3),
    circle(P1, P2, P3, _, _, Radius),
    linear(Angle, Radius, 0.043, -19.442).
```

The *circle* predicate recognises that $P1$, $P2$ and $P3$ fit a circle of radius, *Radius* and *regression* finds a linear approximation for the relationship between Radius and Angle which is:

$$Angle = 0.043 \times Radius - 19.442$$

The _ arguments for circle are dont cares which indicate that, for this problem, we are not interested in the centre of the circle.

This example illustrates an ILP system's ability to use background knowledge to generate high-level features that permit the learning system to refer to meaningful components of a flight. Just as we can describe turn as above, we could also apply linear regression to fit a line to a pilot's approach to the runway, thus discovering the glide slope used. In the following section, we describe an alternative method of invoking complex background knowledge.

5.1 Refinement Rules

Cohen (1996) introduced refinement rules as a method for constructing new literals to be added to clauses during a general-to-specific search. In his FLIPPER program, Cohen used a restricted second order theorem prover

to interpret these rules. The advantage of refinement rules is that they can give FLIPPERs users fine control over how background knowledge is applied in order to create new literals to refine a clause. However, the second-order theorem prover is limited to a simple function-free language.

The system described here is a component of *iProlog* (Sammut 1997). This is an ISO compatible Prolog interpreter with a variety of machine learning tools embedded as built-in predicates. Since the full power of Prolog is available, the refinement rules we implement can invoke arbitrary Prolog programs.

Two types of refinement rule may be defined. A *head rule* has the form:

$$\langle A, Pre, Post \rangle$$

where A is a positive literal, Pre is a conjunction of literals and $Post$ is a set of positive literals. A body rule has the form:

$$\langle \leftarrow B, Pre, Post \rangle$$

where B is a positive literal and Pre and Post are as above.

There must only be one head rule. This indicates that A should be used to create the head of the clause being learned, provided that the condition Pre is satisfied. After A has been constructed, the literals in $Post$, are asserted into Prologs database. There may be any number of body rules. The literals generated by these rules can be added to the body of the clause under construction. Literals in the precondition of these rules can invoke any Prolog program.

Suppose we wish to create a saturated clause (Rouveirol & Puget 1990; Sammut 1981, 1986) based on the same data as Srinivasan and Camacho. The left-hand side of the following rule is the template for the head literal.

```
roll_angle(Radius, Angle)
    where
        true.
```

The *where* part of the rule is the precondition. Refinement rules are invoked in a forward chaining manner. The head rule matches an example

fact, say, roll_angle(1000, 2). Since there are no preconditions, the head of the new clause is created.

The refinement rules for body literals are as follows:

```
:- pos(P, T)
    where
        pos(P, T)
    asserting
        time(T).

:- T1 < T2
    where
        time(T1),
        time(T2).

:- circle(P1, P2, P3, X, Y, Radius)
    where
        pos(P1, _),
        pos(P2, _),
        pos(P3, _),
        P1 \= P2, P1 \= P3, P2 \= P3.

:- Angle is M * Radius + C
    where
        roll_angle(Radius, Angle),
        coefficients(M, C).

coefficients(M, C) :-
    findall(X, pos(point(X, Y, Z), T), Xlist),
    findall(Y, pos(point(X, Y, Z), T), Ylist),
    regression(Ylist, Xlist, M, C).
```

The first rule introduces the *pos* literal. That is, a literal of the form $pos(P, T)$ is introduced into the clause if there is a corresponding fact in the example database. After creating the literal, the postcondition is $time(T)$. This is useful as a typing mechanism for later refinement rules.

The *time* predicate is used by the next refinement rule. This introduces the $before$ literal. In this case, we simply use numeric *less than* to represent

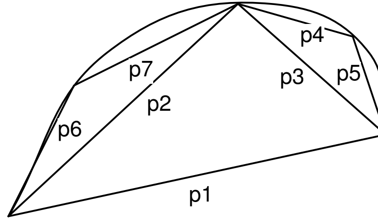


Figure 8: Polygonal approximation of a trajectory

$before$. The assertion from introducing the *pos* literal ensures that, in this case, only comparisons between times are permitted.

We assume that predicates for *circle* and *regression* have already been defined. the *circle* literal in introduced if there are three distinct position facts in the example database.

The final refinement rule introduces a linear relation between roll angle and radius. Note that the preconditions invoke a call to the regression program. The *coefficients* predicate collects the X and Y values of the aircraft's position and passes the lists to the regression program. Again, we have left out error terms to simplify the discussion.

This refinement rule mechanism is implemented in *iProlog* and can be used, as Cohen originally intended, that is to generate literals for a general-to-specific search. They can also be used to produce a saturated clause to be used in a specific-to-general search. This is the manner in which they are currently used. The thing to note is that refinement rules provide a mechanism for invoking quite complex background knowledge.

5.2 Recognising Trajectories

Geometric shapes such as circles and lines are suitable for simple trajectories like turns and climbs, but very often trajectories are much more complicated and therefore more difficult to describe and match. Pearce and Caelli (1997) have devised an instance-based learning algorithm for recognising trajectories.

The first step in their algorithm is to fit a polygonal approximation to a curve (Figure 8).

The system then extracts relations between the lines fitted to the curve.

For example,

$$angle(p_2, p_3, 92)$$

indicates the angle between two of the lines. Each instance of a trajectory is stored in the system's database. Identification of mew trajectories is performed by a constrained graph matching algorithm that is capable of handling relations.

Because of the flexibility offered by the refinement rules described in earlier. It is possible to include this kind of case-based matching as background knowledge. This, where we previously had a *circle* predicate for identifying a circular trajectory, we can also have a more sophisticated matching algorithm for irregular trajectories.

6 Combining Machine Learning and Advice Taking

Although many skills are performed subconsciously, it may still be possible to verbalise some aspects of the skill. For example, as well as acquiring the basic motor skills for controlling an aircraft at a particular instant, pilots must also learn to to plan flights, to navigate according to the plan, they must learn about way points and landmarks, etc. Thus, while the low-level skills that a pilot employs may not be available to introspection, higher-level tasks may be. It is therefore reasonable to acquire behaviours be a mixed strategy of machine learning and advice taking.

Shiraz (Shiraz & Sammut, 1997) has developed a knowledge acquisition system for piloting aircraft in a flight simulator that combines the interactive method of Compton's Ripple-down Rules (Compton & Jansen, 1988) with a machine learning algorithm. Shiraz's system, called *Parvaz*, behaves as follows:

- The autopilot flies the aircraft.
- If the aircraft does not follow the desired trajectory, the human trainer can intervene in either of two ways:
  1. The trainer may enter a rule editing environment, permitting new control rules to be constructed or

2. the trainer may taken over the flight and provide examples of the correct behaviour.

We will briefly describe ripple-down rules before reviewing Shiraz's work.

6.1 Ripple-down Rules

The basic form of a ripple-down rule is as follows:

if condition **then** conclusion **because** case **except**
    if condition **then** conclusion **because** case **except**
        if ...
else **if** ...

Initially an RDR may consist of the single rule:

if true **then** default conclusion **because** default case

That is, in the absence of any other information, the RDR recommends taking some default action. For example, in a control application it may be to assume everything is normal and to make no changes. If a condition succeeds when it should not, then an exception is added (ie. a nested if-statement). Thus the initial condition is always satisfied so when the do nothing action is inappropriate, an exception is added. If a condition fails when it should succeed, an alternative clause is added (i.e an else-statement). The new condition in the exception or alternative clauses is easy to determine.

With each condition/conclusion pair, RDRs store the cornerstone case, i.e. the case that caused the new condition to be created. When a new cases is incorrectly classified, it is compared with the cornerstone case of the incorrect condition and the differences are used to construct the new condition. Usually, the difference list is presented to the expert so that he or she may select the most relevant differences or generalise the conditions. That is, the trainer, never has to explicitly construct a new rule to insert in to the RDR, instead, the knowledge acquisition system present the trainer with two cases and asks which features distinguish the cases. The system then builds the rule and adds it into the appropriate place in the RDR.

6.2 Parvaz

A ripple-down rule system has been added to the same flight simulator that was used by Sammut et al (1992) in their "Learning to Fly" experiments. Four RDR's are used to control each of the four control actions. Initially each RDR consists of the default rule described above. That is, do nothing unless circumstances warrant an action.

Starting on the runway, the aircraft will do nothing, so the trainer intervenes and creates a rule to increase the throttle to 100%. This rule will cause the plane to travel down the runway, but when it does not lift off because no flaps have been applied and the stick has not been pulled back, the trainer again intervenes. The aircraft will then continue to climb. When it fails to level out, the trainer must provide further advice to the autopilot.

In each intervention, the system displays to the trainer the instrument readings at the time the flight was paused. It also displays the readings for the situation that caused the currently active rule to be created. By indicating the significant differences between the two sets of readings, the trainer assists the RDR system in building a new rule.

Shiraz tested the system by asking several subjects to building autopilots for the same flight plan as defined by Sammut et al (1992). The subjects were able to construct rules "manually" for most of the flight. However, some subjects found it was easier, in particularly difficult parts of the flight, to simply take over control and provide examples of the appropriate actions. That is, many stages of the flight are sufficiently simple that control rules can be easily verbalised, however, actions performed in other stages, especially landing, are much more difficult to describe and so teaching by example becomes easier.

In the next section we describe Shiraz's learning algorithm.

6.3 Learning Ripple-down Rules

The learning algorithm also builds ripple-down rules. This permits the automated learner to extend RDR's built manually and *vice versa*. When the autopilot makes a mistake, the trainer provides a single trace of the correct behaviour. The data are segmented and preprocessed just as in Sammut et al (1992). The system executes the RDR's for each control action on the trace data, where the RDR's conclusion differs from the action taken by the

trainer, a new rule is added to the RDR to correct the error. The method for creating a new rule is now described.

Attributes are assigned a priority for each action in order to implement a heuristic to limit the number of conditions in a rule. Initially, all attributes may be given equal priority. For each attribute in the priority list, the algorithm compares the attribute's previous direction with its next direction. If there is a change in direction (e.g. it was increasing and becomes steady) then:

1. Create a test for the attribute. The test is based on the attribute's current value and its previous direction. The test always has the form:

   attribute **op** *value*

   where op is "$\geq$" if the previous direction was increasing and "$\leq$" if it was decreasing. *Value* is the value in the current record.

2. If the new test succeeds for the new case and fails for the cornerstone case, the test is added to the new rule, otherwise the test is discarded.

3. Increment the attribute's priority.

4. If the number of tests in the condition reaches a user defined maximum, scan the rest of the attributes and just update their priorities if their direction has changed. The maximum was set to 3 for these experiments.

Intuitively, the priority reflects a causal relationship between a variable and an action and is related to Stirling's influence matrix.

Shiraz found that all of his subjects were able to construct working controllers by a combination of learning from behavioural traces and and ripple-down rule's semi-automatic knowledge acquisition method. His subjects ranged from novices who were not previously familiar with flight simulators or RDR's to those skilled in flying game simulators and who understood RDR's.

7 Discussion

In this paper, we have reviewed recent research in the application of symbolic machine learning techniques to the problem of automatically building

controllers for dynamic systems. We have shown that by capturing traces of human behaviour in such tasks, it is possible to build controllers that are efficient and robust. This type of learning has been applied in a variety of domains including the control of chemical processes, manufacturing, scheduling, and autopiloting of diverse apparatus including aircraft and cranes.

Recent extensions to the original formulation of behavioural cloning have the common theme of adding greater structure to the representation of the problem. The style of control achieved can be characterised by Nilsson's term *teleo-reactive*, which means that the controller is goal-directed, taking into account and reacting to the current environment. Further progress in this direction is possible, we believe, by continuing to improve the representations and structures available to the learner, to take greater advantage of the abilities of the trainer.

Acknowledgements. Michael Bain is supported by the Australian Research Council.

8 References

Arentz, D. (1994) *The Effect of Disturbances in Behavioural Cloning.* Computer Engineering Thesis, School of Computer Science and Engineering, University of New South Wales.

Benson, S. (1996) *Learning Action Models For Reactive Autonomous Agents.* PhD Thesis, Dept. of Computer Science, Stanford University.

Benson, S., & Nilsson, N. J. (1995). Reacting, planning and learning in an autonomous agent. In K. Furukawa, D. Michie, & S. Muggleton (Eds.), *Machine Intelligence 14.* Oxford: Oxford University Press.

Cohen, W. W. (1996). Learning to Classify English Text with ILP Methods. In L. De Raedt (Ed.), *Advances in Inductive Logic Programming.* IOS Press, pp. 124-142.

Compton, P. & Jansen, R. (1988). Knowledge in Context: A Strategy for Expert Systems Maintenance. In *Proceedings of the Australian Artificial Intelligence Conference.*

Kowalski, R. A. (1995). Using meta-logic to reconcile reactive with rational

agents. In K. Apt & F. Turini (Eds.), *Meta-Logic and Logic Programming.* MIT Press.

Michie, D. (1986). The superarticulacy phenomenon in the context of software manufacture. *Proceedings of the Royal Society of London*, **A 405**, 185-212. Reproduced in D. Partridge and Y. Wilks (1992) *The Foundations of Artificial Intelligence*, Cambridge University Press, pp 411-439.

Michie, D., Bain, M., & Hayes-Michie, J. E. (1990). Cognitive models from subcognitive skills. In M. Grimble, S. McGhee, & P. Mowforth (Eds.), *Knowledge-base Systems in Industrial Control.* Peter Peregrinus.

Michie, D., & Camacho, R. (1994). Building symbolic representations of intuitive real-time skill from performance data. In K. Furukawa, D. Michie, & S. Muggleton (Eds.), *Machine Intelligence 13.* (pp. 385-418). Oxford: The Clarendon Press, OUP.

Michie, D., (1995). Building symbolic representations of intuitive real-time skill from performance data. In S. Wrobel, N.Lavrac, (Eds.), *ECML95.* Berlin: Springer.

Muggleton, S. H. (1995). Inverse Entailment and Progol. *New Generation Computing*, **13**, 245-286.

Nilsson, N. J. (1994). Teleo-Reactive programs for agent control. *Journal of Artificial Intelligence Research*, **1**, 139-158.

Pearce, A. & Caelli, T. (1995). On the efficiency of spatial matching. In *Proceedings of the Second Asian Conference on Computer Visions.* pp. 79-82. Singapore.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* San Mateo, CA: Morgan Kaufmann.

Rouveirol, C., & Puget, J-F. (1990). Beyond Inversion of Resolution. In *Proceedings of the Seventh International Conference on Machine Learning.* Morgan Kaufmann.

Sammut, C. (1981). Concept Learning by Experiment. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.* Vancouver.

Sammut, C. & Banerji, R. (1986). Learning Concepts by Asking Questions. In R. S. Michalski, J. G. Carbonell & T. M. Mitchell (Eds.). *Machine Learning: An Artificial Intelligence Approach, Vol 2.* pp 167-192. Los Altos, California: Morgan Kaufmann.

Sammut, C., Hurst, S., Kedzier, D., & Michie, D. (1992). Learning to fly. In D. Sleeman & P. Edwards (Ed.), *Proceedings of the Ninth International Conference on Machine Learning*, Aberdeen: Morgan Kaufmann.

Sammut, C. (1997). Using Background Knowledge to Build Multistrategy Learners. *Machine Learning* **27**, 241-257.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable domains. In *Proceedings of IJCAI-87*. San Francisco: Morgan Kaufmann.

Shiraz, G. M. & Sammut, C. (1997). Combining Knowledge Acquisition and Machine Learning to Control Dynamic Systems. In M.E. Pollack (Ed.) *Proceedings of the International Joint Conference in Machine Learning*. pp. 908-913. Nagoya: Morgan Kaufmann.

Srinivasan, A. & Camacho, R. (1998). Inductive Logic Programming Applied to an Area of Flight Control. In S. Muggleton, K. Furakwa & D. Michie (Eds.), *Machine Intelligence 15*. Oxford University Press.

Stirling, D., & Sevinc, S. (1992). Automated operation of complex machinery using plans extracted from numerical models: Towards adaptive control of a stainless steel cold rolling mill. In *Proceedings of the 5th Australian Joint Conference on Artificial Intelligence*,

Stirling, D. (1995) *CHURPs: Compressed Heuristic Universal Reaction Planners*. Ph.D. Thesis, University of Sydney.

Urbančič, T., & Bratko, I. (1994). Reconstructing human skill with machine learning. In A. Cohn (Ed.), *Proceedings of the 11th European Conference on Artificial Intelligence*, John Wiley & Sons.