Iranian Journal of Science & Technology, Transaction B, Engineering, Vol. 32, No. B3, pp 295-304
Printed in The Islamic Republic of Iran, 2008

ACQUIRING CONTROL KNOWLEDGE FROM EXAMPLES USING RIPPLE-DOWN RULES AND MACHINE LEARNING*

H. SHIRAZI[1**] AND C. A. SAMMUT[2]

[1]Faculty of Information Technology, School of Computer Science and Engineering,
Malek-Ashtar University of Technology, Tehran, I. R. of Iran
Email: shirazi@mut.ac.ir
[2]Dept. of Artificial Intelligence, School of Computer Science and Engineering,
The University of New South Wales, Sydney, 2052 NSW, Australia

Abstract– The inability of experts to articulate the knowledge required to solve a problem is, arguably, the greatest challenge to building an expert system. The problem is made worse in situations where the response of the expert must be so rapid that there is not even a chance of a plausible *post hoc* reconstruction of the decision processes involved. For this reason, construction of the knowledge base by example is the only approach available. Examples can be used in two ways. They may be used as input to an induction program whose task is to find an abstraction of a control strategy from the data. Examples may also be used to induce the expert to discern differences between cases, thereby allowing the knowledge acquisition system to construct rules semi-automatically. The work presented in this paper demonstrates the feasibility of both approaches. In particular, it shows that the RDR methodology can be extended to domains where the expertise involved is necessarily subcognitive. This is demonstrated by the application of a combination of ripple-down rules and machine learning to the task of acquiring piloting skills for an aircraft in a flight simulator.

Keywords– Machine learning, expert systems, knowledge acquisition, RDR

1. INTRODUCTION

To control a physical process by classical methods, it is necessary to construct a model of the system. Unfortunately, in many physical systems, it is often difficult or impossible to construct an accurate model. Two alternative approaches to controlling physical processes are: using qualitative reasoning [1, 2] and by learning from experience [3-5]. Although qualitative reasoning has shown some promise in some problem domains [6-8], learning from experience has received more attention recently [9, 10]. Here, the system can learn by emulating the behaviour of a skilled operator, that is, learning by observing a human operator, or it may perform trials repeatedly until a given success criterion is met, that is, learning by trial-and-error.

Learning symbolic rules by observing a human operator is also called *behavioral cloning* [11, 12] and has two stages. First, an operator skilled in a task, is asked to control the system. During his or her performance, the state of the system along with the operator's action, are logged to a file. In the next stage, a learning program uses the logged information to construct control rules.

Sammut *et al*, [3] demonstrated a particularly difficult application of behavioral cloning in learning to control an aircraft in a flight simulator. They used a flight simulator on a Silicon Graphic computer and restricted the task to learning to fly a Cessna 150 on a predefined flight path. Three human pilots were asked to fly the aircraft thirty times through the same flight plan. The accumulated file for the thirty flights

*Received by the editors November 4, 2006; final revised form January 5, 2008.
**Corresponding author

for each pilot was used to create a controller for each of the four controls available, namely, elevators, ailerons, thrust, and flaps. Quinlan's C4.5 [13, 14] was used as the induction engine.

Although able to successfully construct an autopilot capable of flying the aircraft through the required flight plan, the original learning to fly experiments encountered a number of problems:

- The decision trees constructed by C4.5 were so large that they were unreadable.
- The autopilots were brittle. That is, they failed when a moderate amount of variation was introduced into the simulation.
- They required a large number of examples from the trainer.

Sammut [12] describes the measures that have been taken to reduce these problems in behavioral cloning. Another alternative to improving the quality of the autopilot is to move away from the fully automatic approach taken by behavioral cloning and to use a semi-automatic learning tool such as ripple-down rules [15].

This paper describes just such an approach. We show that ripple-down rules (RDR) can be used to construct a controller for a complex dynamic system, such as an aircraft. This is significant in that RDR have mostly been demonstrated for classification tasks. We also show that relatively compact and understandable rules can be built without an excessive number of examples being represented by the trainer and these rules can be quite robust. Another feature of this system is that it allows machine learning methods to be mixed with RDR's, drawing on the advantages of each approach.

In the following sections, we briefly describe the problem domain and the nature of the data available. We give an overview of the knowledge acquisition system and give details of the machine learning and RDR tools used. We then report the results of the experiment and conclude with some indications of future research directions.

2. THE LEARNING TASK

The experimental setup for this work is similar to that described by Sammut *et al* [3]. Each pilot is asked to fly a Cessna 150 through a predefined flight plan.

1. A flight starts with the aircraft at the beginning of the runway pointing to the North. It must take off and fly to an altitude of 2000 feet,
2. Then maintain a straight and level flight until a distance of 32000 feet from the starting point.
3. The aircraft then turns right to a compass heading of about 330 and maintains that heading until a distance of about 42000 feet.
4. After reaching that distance from the starting point, the aircraft turns left until it is pointing back towards the runway.
5. The pilot lines up on the runway,
6. descends and
7. lands.

Each stage of a flight needs its own strategy because the pilot's goals at different points in a flight require different actions. Like the original learning to fly experiments, we divide the flight into the same seven stages that we used to specify the flight plan for the pilots.

The source code of the flight simulator has been modified so that,

- the pilot can trace a flight and diagnose incorrect decisions using RDR's;
- RDR's also allow the pilot to add new rules during a flight;
- the display panel has been altered to display some additional variables digitally;
- all the state variables and control settings can be logged to a file.

3. KNOWLEDGE ACQUISITION WITH DRDR

The original motivation for embarking on research in behavioral cloning arose from the observation that many human skills are performed subconsciously. In such cases, it is pointless to ask an expert "how did you do that?" Since that skill is not accessible by introspection, one method for reconstructing the skill is to record its performance and analyze the data with machine learning tools.

The same inability to introspect motivated the development of ripple-down rules. However, the RDR methodology does not seek to entirely exclude the human from the process of building rules to emulate the desired skill. Rather, the human is asked to critique the performance of the rules and point out differences between cases when the program fails. In the case of controlling a dynamic system, RDR's make use of the fact that, while the human operator is not aware of the mechanisms involved in a low-level skill, he or she may be aware of goals and sub-goals. This knowledge allows the operator to reason about cases.

Dynamic Ripple Down Rules (DRDR) implement the RDR method for controlling dynamic systems [16]. The basic algorithm of DRDR is the same as RDR. In actual implementation, DRDR stores the knowledge base as a binary tree with a rule at each node. Each node has a rule condition and conclusion and two branches depending on whether the rule is satisfied or not by the data being considered.

A ripple-down rule as the general form:

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

if ...

else if ...

For example:

if *a* **and** *b* **then** *c* **because** *1* except
if *d* **then** *e* **because** *2*
else if *f* **and** *g* **then** *h* **because** *3*

This is interpreted as if *a* and *b* are true then we conclude *c* unless *d* is true. In that case, we conclude *e*. We will refer to the entire if-then-except-else structure as the RDR and an if-then pair as a single rule.

The numbers, *1*, *2* and *3*, in this example, refer to cases. RDR's are built incrementally. Each time the RDR fails to produce the correct answer, a new rule, along with the case that caused its creation, is added to the RDR. Suppose an RDR fails. We compare the new case with the case associated with the last fired rule. The differences give rise to the conditions that will be placed in a new rule that will distinguish the new case from the old. If a rule's conditions were satisfied by a case when they should not have been, a new rule is added as an *exception*. When a rule's conditions were not satisfied by a new case when they should have been, a new rule is added as an *alternative* (else).

The initial RDR usually has the form:

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 not make any changes. If a condition succeeds when it should not, then an exception is added (i.e. a nested if-statement). Thus the initial condition is always satisfied, so that when the *do nothing* action is inappropriate, an exception is added.

A ripple-down rule can also be viewed graphically as a binary tree in which nodes are condition/conclusion pairs and the exceptions and alternatives follow the branches of the tree.

Although the basic RDR methodology has been retained, we require additional features in order to apply ripple-down rules in this domain. These extensions form the basis of Dynamic RDR's (DRDR), and are described next.

The pilot can pause a flight to investigate the DRDR when it is not flying according to the pilot's wishes. When this happens information must be transferred from the flight simulator to the knowledge acquisition tool, where much of the information is presented graphically.

a) Dealing with multiple actions

In this experiment, we required a system capable of producing four conclusions given a simulator state (one for each of the elevators, flaps, ailerons, and throttle). One way to solve this problem is to use Multiple Classification RDR instead of simple RDR [18]. However, we found it much easier to use a system that can handle four separate knowledge bases concurrently.

DRDR handles the four knowledge bases and outputs multiple conclusions (one from each of the four RDR's). Whenever the pilot pauses the flight to investigate the current rules, he or she is able to see DRDR's conclusion for all action variables.

b) Creating rules by actions

When creating new rules for elevators or ailerons, it is difficult for users to provide the exact value for their conclusion. It is easier for them to show the correct action by moving the joystick. In the present implementation, it is not possible to access the joystick to obtain the control values, however, an approximation to the joystick is provided by analog slider controls

c) Interaction between DRDR and LDRDR

As we shall see, the use of RDR's still requires the pilot to reason about his or her actions. Although the RDR methodology makes this much easier, there are still some cases when it is easier to show the knowledge acquisition system more examples and let it induce the appropriate rules. Thus, DRDR has the ability to interact with a learning program that also generates ripple-down rules.

4. LEARNING RULES USING LDRDR

For fast dynamic systems, experiments have shown that it is very difficult for an expert to completely describe his or her strategy and the reasons for choosing that strategy [4, 17]. Further, the descriptions are incomplete and approximate and cannot be directly translated into an automatic controller. Nevertheless, such descriptions contain general information about the system and they can be used as guidelines for constructing an automatic or semi-automatic controller. Therefore, it is attractive to attempt to modify and complement manually created rules by rules that have been automatically created from pilot performances.

To automatically construct rules from the pilot's behavior, we use LDRDR [16]. The basic algorithm for LDRDR is to search the data logged from a flight, record by record, and find those attributes that cause an action, then create rules based on those attributes and their qualitative states. The assumption is that the qualitative state of variables changes when an action is performed.

The main reason for introducing LDRDR [16] instead of using one of the existing machine learning programs was the need to deal with sequential data. In addition, it was necessary to have a program that was compatible with DRDR, with the ability to learn incrementally. Other machine learning algorithms that construct ripple-down rules, for example, Induct/RDR [15], are batch algorithms and are not designed to deal with sequential data. LDRDR is specifically designed to work with sequential data. It takes the current knowledge base, the behavioral traces, and a priority list as its input. It creates rules that are added

to the knowledge base to deal with cases not previously handled correctly. The LDRDR algorithm constructs a controller as follows:

for each record in the behavioral trace:
- Test the next record against the knowledge base;
- If the conclusion of the RDR differs from the recorded trace, create a new rule to correctly handle the new record;
- Add the new rule to the knowledge base.

The condition part of a new rule is constructed by examining those variables that change most in correspondence with changes in control actions. The behavioural trace includes all the information that a pilot would normally see in the aircraft instruments. LDRDR tries to relate the pilot's actions to changes in the instrument readings, and thereby predicts what actions are required depending on the state of the instruments.

The logged data usually contains information from different stages. This data is usually noisy and contains many redundant records. Pre-processing prepares the logs for the learning program. This includes segmenting the logged data, discretising control values, eliminating spurious values and creating separate inputs for each control action.

After pre-processing, each of the data files and the existing RDR's for a particular control action are to the LDRDR algorithm. LDRDR also uses a priority list, which we discuss later. The output of LDRDR is an extension of the original RDR to cover cases in the input data that were not covered by the original RDR. The new RDR is converted into C if-statements by recursively traversing the RDR and creating one if-statement for each rule. An if-statement's conditions are the conjunctions of all true conditions in the RDR from the root to the rule. Rules will be executed sequentially. If any of the rules is executed, control will jump to the end of the list to guarantee a single conclusion from the tree. If none of the rules fires, the last rule, which is the default rule, will be executed.

a) Data collection and preparation

For those parts of the flight where it is difficult for the pilot to diagnose a problem or suggest correct rules, he or she can simply show the correct action by switching to manual mode and flying the aircraft. During the flight, the pilot's actions, along with the state of the simulation are logged to a data file. The state information is logged every second or when a change to a control action has been detected.

There is always a delay between a stimulus and a response. Ideally we would like to record the pilot's action and the event that actually provoked the action. The problem is how can we know when that was? This is not a trivial issue, because human reaction time is not a constant, varying from person to person. Moreover, the reaction time of a person varies depending on the task. If the task is performed frequently and becomes familiar, the response time is shorter than for situations which are new for the human. In addition to these problems, pilots usually anticipate the future location of the aircraft and prepare a response for that state. In this experiment, following Sammut, *et al* [12], we decided to use a response time of one second in each stage of the flight. We have not attempted to model the pilot's predictive behavior.

As with DRDR, different sets of rules are created for each stage. In each stage, different knowledge bases are created for each action. Therefore, twenty-eight knowledge bases have to be created for a complete flight (4 control action, 7 flight stages).

The first stage of pre-processing is to segment the data into the stages where they were recorded. To make the segmentation of logged data easy, a new attribute has been added at the beginning of the recorded data to show the record's stage number. Based on this attribute, the filtering program segments the recorded data.

The Cessna aircraft's control system consists of four moveable control surfaces (elevator, flaps, ailerons, and rudder) plus the throttle. The values of the control variables are continuous. However, the learning algorithm can only deal with discrete class values. To solve this problem, a pre-processor is used to sub-divide the variable settings into intervals that can be given discrete labels. The range for each partition is chosen by analyzing the frequency of the occurrence of the values. The method of discretisation follows Sammut et al. [12].

b) Constructing new rules

After performing the pre-processing just described, each of the data files and the existing RDR for a particular control action are used to extend the RDR using the LDRDR algorithm. LDRDR also uses a priority list. This is a list of attributes which are sorted based on their importance relative to the action. There is a priority list for each control action. The priority lists are described in the next section in more detail.

The algorithm extends RDR as follows:

Inputs: current knowledge base; behavioural traces; a priority list.

for each attribute in the priority list:

1. Compare the attribute's previous qualitative state with its next qualitative state. If there is a change (e.g. it was increasing and becomes steady) then:
2. Create a test for the attribute. The test is based on the attribute's current value and its previous qualitative state. The test always has the form:

   attribute **op** value

   Where **op** is ">=" if the previous direction was increasing and "<=" if it was decreasing. "*Value"* is the value in the current record. The new test is applied to the cornerstone case associated with the last rule that was satisfied to make sure the test is true for the current case but excludes the cornerstone case.
3. Add the test into a condition list.
4. Attributes in the priority list are ordered by a numerical score. Increment the attribute's priority.
5. If the number of tests in the condition list reaches a user defined maximum, scan the rest of the attributes and simply update their priorities if their qualitative state has changed.

end loop

If the condition list is not empty, create a rule and add it to the RDR. The conclusion of the rule comes from the action recorded in the trace. The current record becomes the rule's cornerstone case. The rule will be added as an exception to the last rule in the RDR if that rule is evaluated true (true branch). It is added as an alternative if false (else).

The output of LDRDR is an extension of the original RDR to cover cases in the input data that were not covered by the original RDR.

c) The priority list

A human expert usually only considers a small number of variables at any time [15]. To emulate this behaviour, LDRDR limits the number of conditions per rule. To avoid missing important conditions in rule generation, LDRDR creates a priority list for the actions. The expert can provide the priority list or LDRDR can use its default priority list. If LDRDR creates the priority list, the priority of all attributes will be initialized to zero at the beginning of learning. If the expert decides to enter his or her own priority list, he or she can specify a set of attributes that will be tested first and the rest of the attributes will be decided by LDRDR. The expert also has the opportunity to enter a list of attributes in order and let LDRDR update the list automatically.

There is a priority list for each control action. These lists can be transferred from one stage to another. During learning, the priority list is updated automatically by considering attributes that contribute more in rule generation (except for the attributes that the expert decides are to be tested first). The priority of an attribute increments by one each time LDRDR notices a change in its qualitative state. This list is always sorted by priority.

LDRDR chooses attributes from the top of the list. After choosing the attribute, LDRDR looks at the logged data to decide whether there is a change in the qualitative state of that attribute or not. The attribute will be included in one of the tests in the new rule if there is a change in its qualitative state. This process continues until LDRDR creates the maximum number of tests allowed or reaches the end of the priority list.

5. CONSTRUCTING AN AUTOPILOT WITH PARVAZ

If autopilot rules can be built by induction from examples why do you need any other method of knowledge acquisition? Although better suited than C4.5 for building control rules for dynamic systems, LDRDR still suffers from the problem that the rules that it synthesises can be more complex than they need to be. As we mentioned earlier, the main reason for invoking LDRDR is when the pilot finds it too difficult to reason about his actions, usually in situations that require rapid responses.

The manual knowledge acquisition part of this system (DRDR) is effective when it is possible for the pilot to articulate rules related to his or her performance. These rules tend to be general in nature and if they are not complete in themselves, they are often useful as constrains on the learning system. Thus, the complexity of the rules is reduced.

Knowledge acquisition begins by applying some simple rules created manually by DRDR or by using LDRDR to logged information. In both cases, rules can be tested by running the simulation in autopilot mode. During the flight, if the aircraft does not follow the expected path, the pilot is able to pause the flight and trace the rules that have been executed or are currently being executed. The pilot is also able to modify existing rules that seem incorrect by adding new rules. In this case, the previous and current status of the aircraft, plus all the simulations state variables, will be presented to the pilot. Also, all the rules under execution for each control action will be reported.

If the pilot decides to create rules using LDRDR, he or she must fly the aircraft and record data. This task can be repeated as many times as the pilot wants. After logging data, the log files are preprocessed and passed to LDRDR

The above procedure is repeated until a set of successful clones has been created. In our experiments, knowledge bases are created one stage at a time. This has the advantage that rules constructed for one stage can be transferred and adapted for other stages, thus reducing some of the effort required in later stages. The priority lists can also be transferred from one stage to another or new priority lists can be created for each new stage.

After every modification of the RDR (manually or automatically), the flight simulation is run in autopilot mode to test the new RDR. To do this, the code of the original autopilot is replaced by the RDR (translated into C). A C function is also incorporated into the flight simulator to determine the current stage of the flight and when to change stages. The appropriate set of rules for each stage is then selected from four independent if-statements in each stage for every control action [3].

a) Analyzing rules

Information about the knowledge structure in the system is presented graphically to assist the pilot in understanding the current state of the knowledge base. The pilot may view the RDR either in text form as

rules or graphically as a tree structure drawn on the monitor. The tree may be traversed by moving the mouse over the tree.

1. Drawing the trees and interpreting them as if-then rules: Drawing the RDR's makes it easy for the pilot to examine the rules from the root to each node. This option also provides the pilot with a feeling for the size of the RDR and the number of refinements that he or she has made so far. However, just by looking at the RDR, it is difficult to get a detailed view of the knowledge bases or their complexity.

DRDR has the option to convert the RDR into a decision list of if-then rules. In this case, each node of a RDR will be displayed as a rule. This includes all the satisfied conditions from the root to that node. In our experiments we found this option very useful for users when they needed to create new rules. Rules are stored as C if-then statements. A macro, "THEN" has been added for readability.

2. Navigating through the tree and investigating rules: To investigate the rules for each control action or to create a new rule for an action, the user must have knowledge not only about the state of the aircraft, but also about the rules. However, displaying the rules or drawing the RDR does not provide sufficient information for the user to understand what has happened so far or why the rules do not work properly. DRDR has to provide more information to make the context useful to the user.

DRDR makes it possible for the pilot to investigate each node by entering its number. In this case, the current status of the aircraft plus the conditions satisfying the last rule are presented. Also, the conclusion reached and all the satisfied conditions are reported to the pilot. The rule's number plus the number of its parent and children are also available. The pilot is also able to see the information about all these nodes.

b) Creating new rules

After evaluating the existing rules, if the pilot finds a conclusion is wrong or there is no interpretation for the current situation then he or she is able to add new rules. To help, the system provides a list of differences between the current state of the flight and the state of the simulation associated with the last condition that was satisfied. Choosing one or more variables from this list guarantees the production of a rule that will correctly interpret the new case but not the old one.

c) Logging flight information

For those parts of the flight where it is difficult for the pilot to diagnose a problem or suggest correct rules, he or she can simply show the correct action by switching to manual mode and flying the aircraft. During the flight, the pilot's actions, along with the flight information are logged to a data file. The flight's information is logged every second or when a change in a control action has been detected. These data can then be used to construct rules using LDRDR.

6. TESTING THE SYSTEM

Ideally, we would like to test Parvaz using real pilots and a more realistic flight simulator. Lack of access to real pilots and a higher quality flight simulator made this impossible. However, since the aim is to investigate the possibility of transferring an expert's knowledge into an automatic controller using Parvaz, any user who can use the flight simulator to fly the aircraft and land successfully can be considered to have some level of expertise.

Parvaz was tested using three volunteers. Among the subjects, only one of them, the first author, was familiar with Parvaz, while another was familiar with a flight simulator. All subjects were postgraduate students in computer science. Their task was to create a set of knowledge bases using Parvaz that could successfully complete the previously specified flight plan.

Prior to the experiments, the subjects received a one hour tutorial in the use of Parvaz and attended a demonstration. During the demonstration, the demonstrator explained how he flew the aircraft and the subjects were allowed to ask questions about the flight and the flight plan. They were allowed to practice with the simulator until they became proficient in flying the aircraft. During the experiments the following data were collected:

- the number of times that subjects interrupted the flight simulator to modify the knowledge base or investigate it,
- the number of rules they created,
- the frequency of use of DRDR and LDRDR,
- the amount of time they spent creating rules,
- their comments about Parvaz.

All the subjects were successful in creating a set of rules that could fly the aircraft through the given flight plan. The size of the knowledge bases varied considerably from one subject to another one. One subject took care to create rules for any possible state of the aircraft and thus produced a large RDR. Another subject was only interested in building a set of rules that could fly the specific plan and so produced a much smaller RDR. An interesting observation during these experiments was the reuse of knowledge. All the subjects tried to use rules created in an earlier stage if the task was similar. This was especially true when creating rules for the elevators in stages three, four and five, where the rules created for stage two were reused and modified by some exception rules. Table 1 shows a summary of these data.

Table 1. Summary of the experiments

	Subject 1				Subject 2				Subject 3			
	P	D	L	T	P	D	L	T	P	D	L	T
Stage 1	13	13	0	13	5	13	0	13	5	10	0	10
Stage 2	8	10	10	20	7	14	5	19	4	8	3	11
Stage 3	9	13	19	32	20	22	10	32	5	11	5	16
Stage 4	8	20	19	39	7	17	5	22	3	12	0	12
Stage 5	16	13	26	39	38	44	44	88	5	14	2	16
Stage 6	21	20	27	47	21	19	0	19	17	19	0	19
Stage 7	20	19	30	49	19	16	0	16	6	10	3	13

P: number of pauses, D: number of DRDR rules, L: number of LDRDR rules, T: Total number of rules

7. CONCLUSION

The work presented in this paper demonstrates the feasibility of both uses of machine learning and knowledge acquisition methods in learning to control a dynamic system. In particular, it shows that the RDR methodology can be extended to domains where the expertise involved is necessarily subcognitive. This is demonstrated by the application of a combination of ripple-down rules and machine learning to the task of acquiring piloting skills for an aircraft in a flight simulator. It also showed a solution to overcome the bottleneck of knowledge acquisition in some dynamic domains which involves subcognitive skills.

There are several directions in which future research might be continued.

- It was noted that the *pilots* in these experiments used DRDR as the primary tool for building rules, but when the situation under analysis becomes too complex, LDRDR was invoked. A question of considerable interest arises: is it possible to quantify the point at which the cross-over occurs?
- While RDR's have proved to be effective in capturing the pilot's knowledge, ideally a real pilot should never see them. That is, the pilot's interaction with the knowledge acquisition system should be entirely *within the cockpit*, providing examples and comments just as an instructor would do, by pointing to instruments, etc.

The robustness of LDRDR needs to be investigated further, especially when noise, such as turbulence and wind drift, are introduced into the simulator.

REFERENCES

1. Bratko, I. & Succ, D. (2004). Learning qualitative models. *AI Magazine*, Vol. 24, No. 4, pp. 107 – 119.
2. Srdoc, A., Bratko I. & Sluga, A. (2007). Machine learning applied to quality management-A study in ship repair domain. *Computers in Industry*, Vol. 58, No. 5, pp. 464-473.
3. 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.
4. Urbancic, 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.
5. Kadous, M. W. & Sammut, C. (2005). Classification of multivariate time series and structured data using constructive induction. *Machine Learning*, Vol. 58, pp. 179-216.
6. Monzina, M., Zabkar, J. & Bratko, I. (2007). Argument based machine learning. *Artificial Intelligence*, Vol. 171, No. 10-15, pp. 922-937.
7. Succ D. & Bratko (2000). Problem decomposition for behavioral cloning. *Lecture Notes in Computer Science*, Vol. 1810, pp. 382 – 391.
8. Yik, T. F. & Sammut, C. (2007). Trial-and-error learning of a biped gait constrained by qualitative reasoning. In M. Srinivasan & M. Dunbabin (Eds.), *Australasian Conference on Robotics and Automation*, Brisbane.
9. Sammut, C., Kadous, W. & Sheh, R. (2007). Learning to drive over rough terrain. In K. Furukawa (Ed.), *International Symposium on Skill Science*, Tokyo.
10. Bratko, I. & Succ, D. (2002). Using machine learning to understand operator's skill. *Lecture Notes in Computer Science*, Vol. 2358, pp. 812 – 823.
11. Kadous, W., Sammut C. & Sheh, R. (2006). Autonomous traversal of rough terrain using behavioural cloning. *The 3rd International Conference on Autonomous Robots and Agents*.
12. Sammut, C. (1996). Automatic construction of reactive control systems using symbolic machine learning. *Knowledge Engineering Review*.
13. Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. San Mateo, CA: Morgan Kaufmann.
14. Mulholland, M., Hibbert, D. B., Haddad, P. R. & Sammut, C. (1994). *The application of the C4.5 classifier to building an expert system for ion chromatography*. Chemometrics and Intelligent Laboratory Systems.
15. Gaines, B. R. & Compton, P. (1992). Induction of ripple-down rules. *Proceeding of the 5th Australian Joint Conference on Artificial Intelligence*, Hobert, Tasmania.
16. Shiraz, G. M. & Sammut, C. (1997). Combining knowledge acquisition and machine learning to control dynamic systems. *The Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, Nagoya, Japan.
17. Potts, D. & Sammut, C. (2005). Incremental learning of linear model trees. *Machine Learning*, Vol. 6, No. 1-3, pp. 5-48.
18. Dazeley, R. & Kang, B. (2003). Weighted MCRDR: deriving information about relationships between classifications in mcrdr. *The 16th Australian Joint Conference on Artificial Intelligence* (AI03). 2003. Perth, Australia: Springer-Verlag, Berlin Heidelberg New York.