12

Error Tolerant Learning Systems

C. Sammut†
The Turing Institute,
Glasgow, UK

Abstract

We consider the task of a robot learning in a reactive environment by performing experiments. A reactive environment is one where changes occur in response to actions. Actors other than the learner may be present in the world. The robot performs experiments by modifying the environment and observing the outcome. These observations lead to a collection of concepts which constitute a theory of the behaviour of the environment, also called a world model. An experiment may either increase confidence in a theory or refute a theory, but it can never prove a theory. Therefore it is possible that the robot will develop an inaccurate model of its world. This paper discusses a number of issues involved in finding and repairing faults in a world model. It also describes some preliminary results obtained from a learning program called CAP.

1. INTRODUCTION

Learning systems can be divided into two categories: single-concept learners and incremental learners. AQ (Michalski 1983) and ID3 (Quinlan 1983) are examples of single-concept learning systems. They produce one set of rules from one set of data and have no memory which permits them to add to a knowledge base by further learning. Incremental learning systems remember the concepts which they have learned and can use them for further learning and problem solving. Some examples are, CONFUCIUS (Cohen 1978) and Marvin (Sammut 1981). These programs build a model of their task environment through successive learning experiences which require interaction with the environment.

The task that we consider in this paper involves a program learning to control an agent in a reactive environment. This is an environment where changes occur in response to actions. Agents other than the learner may be present. As an agent accumulates experience, it constructs a world model or theory of behaviour which can be used to predict the outcome

†Present address: Department of Computer Science, University of New South Wales, Sydney, Australia.

of events and to determine what actions may be necessary to solve problems. In our discussion, a theory is a collection of concepts, where a concept is a description of a class of objects or events.

The concepts which an agent can learn are determined by the events experienced by that agent, so any concepts created on the basis of only a small amount of experience are likely to be less accurate than concepts developed on the basis of more extensive data. This is so because a small number of instances of a concept may not contain sufficient information properly to characterize the concept. Because of this, it may seem wise to postpone concept formation until sufficient data are available. This may not be possible for a number of reasons:

1. The agent may be required to perform some task even before it has acquired enough experience to build an accurate model of its world. For example, the system may be responsible for controlling the attitude of a satellite and learning to compensate for changes in atmospheric drag, changes in mass, and so on.
2. Unless the program builds a world model, it has no means of assessing the usefulness of the information it gains. Thus it is impossible to determine when 'sufficient information' for concept formation has been accumulated. The program is also unable to employ a directed search for further data since there are no criteria for evaluating the usefulness of information.

A learning system which postpones theory formation can acquire data either by passively observing the environment and other actors, or it may perform its own, unguided actions. With passive observation, there is no guarantee that the observations will be representative of the world unless an agent is acting as a teacher and showing the learner 'interesting' things. Thus it may be necessary to wait a very long time before sufficient data for theory formation have been accumulated.

Active learning requires the agent to perform experiments, that is to perform actions and note the changes in the world. A random choice of an action is as good as any when there is no theory to guide the learner. Unfortunately, random actions in a complex domain may produce instances of many concepts, giving the learner a confused mass of data. One of the benefits of constructing a partial theory while accumulating data is that the partial theory will suggest what kind of data to look for. If part of the theory can be identified as deficient then a strategy may be devised to seek the information necessary to repair the theory.

For incremental learning systems which operate in reactive environments, the biggest problem is maintaining consistency in a knowledge base which is constantly changing. Observation of the world can cause new concepts to be hypothesized or existing ones to be revised. Thus the world model, which is stored in the knowledge base, evolves over time.

In the remainder of this paper we discuss a number of aspects of knowledge base maintenance, including determining that an error in the world model exists, finding the error, and repairing it. We also discuss some work in this area including Hume's CAP program (Sammut and Hume 1987), and related work.

2. PROBLEMS IN INCREMENTAL LEARNING

Let us now try to be more specific about the nature of the errors that can be made by an incremental learning algorithm. First, because the system is incremental and therefore responsible for building its own background knowledge, we assume that all concepts to be learned must be describable in terms of concepts previously acquired. This assumption leads to two possible errors:

1. The system may observe an event for which there is no prior knowledge that will allow the system to make generalizations and thus learn.
2. The system may observe an event for which there is prior knowledge. However, the known concepts are too general to describe the concept of which this event is an instance.

Thus, an inadequate knowledge base can result in either too specific or too general a concept being learned. Interestingly, Shapiro (1981) notes that analogous errors can occur in pure logic programs (as well as non-termination). Later we will show how some of the techniques proposed for debugging PROLOG programs can be used to 'debug' the knowledge base of a learning system where concepts are represented by Horn clauses in first order logic.

An example of an error introduced into the knowledge base by over-generalization follows. Suppose the system is learning the preconditions for pouring liquids into containers. First it must learn what a suitable container is. This can be done by attempting to pour water onto a number of different objects. The following sequence of actions illustrates some of the pitfalls of learning by experimentation:

1. Water is poured over a closed box. This fails because the water ends up on the floor rather than in the box.
2. Water is poured over a cup and succeeds.
3. Water is poured over an open cylinder and this also succeeds.

What can explain these successes and failures? Any explanation which the learning system attempts must be in terms of what it already knows. If it knows about objects that have circular cross-sections, it may hypothesize that the precondition for pouring a liquid into an object is that the

object must have a circular cross-section, as do the cup and open cylinder. Of course, this theory will fail if the robot tries to pour water into a closed cylinder.

A more useful concept for learning about pouring is that of convex shapes. However, the robot may not have learned this concept before trying to learn about pouring. Therefore, it makes an over-generalization. Why should the robot attempt to learn something when it is not prepared for it? An observation of the world may have brought its attention to this task. For example, one mechanism that children use to explore the world is to imitate adults, adding some variations of their own. Observing what are assumed to be rational actors provides a good focus of attention for learning; unfortunately, the actors may be observed doing something more complicated than the child can understand. That is, it has not yet learned all the background concepts necessary to describe adequately what is seen. This is exactly the situation in the case of the robot learning about containers into which it can pour liquids.

2.1. Recognizing that an error exists

A world model is intended to predict the outcome of events in the world. A theory is clearly incorrect if an unexpected outcome occurs. When the robot pours water over a closed cylinder, it has a theory that predicts that the water will remain in the cylinder. When the water ends up on the floor, there is obviously something wrong. However, recognizing that an error exists and knowing what it is are not the same thing. For example, it is easy to see that a computer program has a bug, but locating the bug is much more difficult.

2.2. Locating errors in a theory

A theory can be thought of as a network of interconnected concepts. The shaded node, *E*, in Figure 1 may represent the concept of an object that can contain liquid. This is needed to establish the preconditions for retaining liquids in some container. This concept may in turn be necessary for knowing how to make a cup of tea, for example. What happens if, in the process of making a cup of tea, the tea is spilt on the table. Which part of the world model was responsible for the mishap?

One way of locating the problem is to trace through the execution of the plan that leads to the unexpected result, testing each concept that contributed to the plan. That is, we debug the plan. In logic programming languages, declarative descriptions can be executed as programs, thus the distinction between a concept and a plan is blurred. This is very helpful for our purpose since, by adopting a Horn clause representation of concepts, we can profit from experience gained in writing intelligent debuggers for PROLOG. Sussman (1973) developed a debugger for

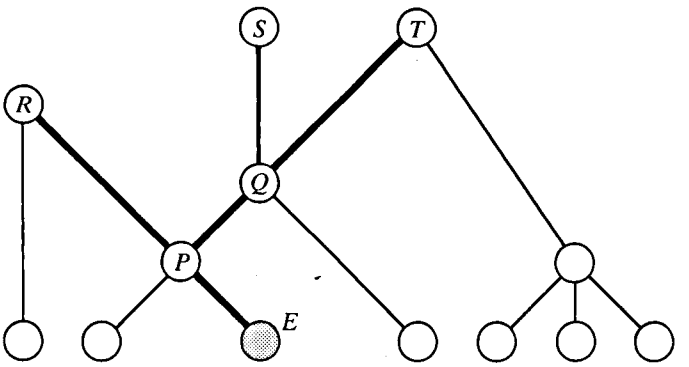

Figure 1. A theory is a network of concepts.

procedural progams. However, his program, called HACKER, used an *ad hoc* approach which required a library of possible fixes to common programming errors. We wish to minimize the number of assumptions necessary to locate errors in a theory. Horn clause logic gives a uniformity of representation which allows us to break errors into only three types.

Shapiro (1981) describes a program, MIS, for detecting logic errors in pure PROLOG programs. He claims that three categories are sufficient to characterize errors in pure logic programs:

(1) the program fails to terminate;

(2) the program returns an incorrect solution;

(3) the program fails to return any solution.

In the case of a faulty theory, return of an incorrect solution corresponds to an over-generalization in the theory since a concept has been used to describe an event that it should not be able to recognize. Failure to return an answer corresponds to a theory that is too specialized since it has failed to recognize an event it should have. Non-termination of a theory can occur when recursive concepts are involved. We will not dwell on the latter, but concentrate instead on theories that are too general or too specific. Note that some concepts in a theory may be too general while others are too specific.

To locate an error when an incorrect solution has been given (that is, the theory contains an over-generalization) Shapiro's debugging algorithm uses a method called *backtracing* to work backwards through the failed proof of a goal, searching for the procedure that caused the failure. In Figure 1, backtracing would begin with the last goal satisfied, that is, *T*. The debugger begins stepping back through the proof, i.e. down the dark path to node *Q*, then *P* if necessary, asking an oracle if the partial solution at each point is correct. If this is not true, then an erroneous

clause has been found. Note that the algorithm assumes the existence of an infallible oracle. In a reactive environment, the learning program can do without an oracle since the program is able to perform experiments to test a concept. Thus a failure suggests that the initial set of experiments which resulted in the formation of the concepts along the solution path was not extensive enough for at least one of the concepts. In the case of making a cup of tea, experimentation may identify the concept that describes preconditions for pouring liquids as faulty.

A concept that is too specific may prevent the program from being able to form a plan to achieve some goal. That is, the logic program that is supposed to satisfy the goal does not cover the initial conditions of the task. An attempt at debugging the theory can only be made when a correct solution has been seen, otherwise the learner has no indication that the task really is possible. A correct solution may be found, either by 'mutating' the current theory in the hope that the goal can be satisfied by the mutant, or by the learner observing another agent in the world performing the task. Shapiro's debugging method for programs that fail to produce an answer is equivalent to the second alternative, that is, the oracle supplies the correct solution. The debugger again tries to work backwards seeking clauses in the program that could have produced the given solution. Once such a clause is found, its body provides further goals that should be satisfied in order to arrive at the solution. The debugger considers each of these intermediate goals to see if they can also be produced by other clauses. Any goal that cannot be achieved indicates where the program or theory is deficient.

It should be noted that more intelligent search methods can be used to reduce the number of nodes tested. But while a method such as Shapiro's is very useful, it assumes that the learning system is able to suspend its current activities while it seeks the error in its theory. However, we noted earlier that there are applications where this is an unaffordable luxury. If the current theory works well in most cases and has failed relatively infrequently, it may be better to defer a thorough attempt at debugging in favour of persisting with the existing theory, while collecting information which will eventually point out the incorrect concept. For example, if over a period of time, all the concepts R, S, and T are noted to have errors then the system may conjecture that all the failures have a common cause and that there are only two concepts which could be that cause. Thus the search for the error in the theory has been reduced at the cost of tolerating more failures. A difficult problem is to decide whether it is more costly to stop and debug or to continue with some failures.

2.3. Correcting errors

After an erroneous concept has been detected what should be done to repair it? In his MIS program, Shapiro reasoned as follows: when a

program has failed to produce an answer, the program is too specific, so to repair it the debugger should generalize the program in order to cover the goals that cannot be satisfied. On the other hand, if a program produces an incorrect answer, i.e. it satisfies a goal it shouldn't, then it is too general and therefore the debugger should make the program more specific. This excludes the goals that were incorrectly satisfied. In order to perform these generalization and specialization operations, the program must have a *refinement operator*, which, given a term, will produce a set of terms that are minimally more specific than the original one. Thus, by recursive application, the refinement operator defines a language that is a subset of Horn clause logic. If MIS is to be able to create a correct program successfully, the refinement operator must be capable of generating the clauses necessary for the correct program. If the refinement operator defines too large a language, then the search time for the required clauses will be prohibitive. If the language is too small, then it may not be possible to generate the required clauses.

What do generalization and specialization mean when objects and events are represented by Horn clauses? A clause such as:

$$X \leftarrow A \wedge B \wedge C \wedge D \wedge E$$

states that an object satisfying $A \wedge B \wedge C \wedge D \wedge E$ belongs to class X. When a clause describes an instance of a concept, all the literals in the clause are ground, that is, they contain no variables. A clause is generalized by replacing constants by variables and by replacing predicates such as A, B, C, etc, that describe some property of the object by other predicates that have more relaxed restrictions on the range of values which the property may assume. Specializing a clause introduces new restrictions on those ranges of values. MIS required that the predicates used in generalization and specialization were generated by a predefined refinement operator. To avoid having to know too much about the problem domain before starting, it is desirable to allow new terms to be introduced as required by the data. Suppose our learning system does not know a concept that will distinguish between a cup and a bowl on one hand and a closed cylinder on the other hand. However, by experimentation it is clear that there is some difference because the first two objects retain liquid poured over them while the closed cylinder does not. The learning system ought to be able to propose a new concept for objects that retain liquid. A method capable of this behaviour was first proposed by Sammut (1981). More recent related work by Muggleton is described below.

Muggleton's DUCE (1987) relies on a set of operators to *compact* the description of a set of examples to a simpler description. Each example is represented by a propositional Horn clause. Some operators preserve the equivalence of descriptions but reduce the number of symbols

required, while others produce generalizations. There are six operators in all and they are the basis for a method of anti-unification. Indeed, one of the goals of this work was to produce a complete inverse of resolution (see this volume). All six operators are necessary for the completeness of the theory, but pairs of operators are sufficient for induction. We will describe one such pair and refer the interested reader to Muggleton's paper (1987) for the complete description. A first-order version of DUCE, called Cigol (logiC backwards), has now been implemented (Muggleton and Buntine 1988).

Absorption. Given a set of clauses, the body of one of which is completely contained in the bodies of the others, such as:

$$X \leftarrow A \wedge B \wedge C \wedge D \wedge E$$
$$Y \leftarrow A \wedge B \wedge C$$

we can hypothesize:

$$X \leftarrow Y \wedge D \wedge E$$
$$Y \leftarrow A \wedge B \wedge C$$

In fact, this is the generalization rule used by Sammut (1981) in his program, Marvin.

Intra-construction. This is the distributive law of Boolean equations. Intra-construction takes a group of rules all having the same head, such as:

$$X \leftarrow B \wedge C \wedge D \wedge E$$
$$X \leftarrow A \wedge B \wedge D \wedge F$$

and replaces them with:

$$X \leftarrow B \wedge D \wedge Z$$
$$Z \leftarrow C \wedge E$$
$$Z \leftarrow A \wedge F$$

Note that intra-construction automatically creates a new term in its attempt to simplify descriptions. At any time during induction there may be a number of applicable operators. The one chosen is the operator that will result in the greatest compaction.

As a robot performs experiments, its experiences may be stored as clauses representing observations. As this collection of clauses grows, it can be compacted, using the DUCE operators. This has the effect not only of reducing the storage cost of the information, but also of detecting patterns and introducing new terms into the knowledge base. For example, DUCE could easily detect the similarity of the cup and bowl when liquid is poured over them. The intra-construction operator would

introduce a new concept, that is, a new set of clauses whose heads have the same principal functor.

2.4. Maintaining consistency

Detecting and repairing an error in a single concept is one thing, but repairing an entire theory is another matter. Remember that in Figure 1, we envisaged a world model or domain theory as a network of interconnected concepts. Using a Horn clause representation, the head of a clause corresponds to a parent node and the goals in the body correspond to the children. These goals match other clause heads and form links to the rest of the network. Also in Figure 1, we imagined that one concept, represented by the shaded node, E, was in error. When the concept is repaired, what effect will that have on the concepts that referred to the old concept? Since P, Q, R, S, and T refer, directly or indirectly, to the erroneous node they must have been learned in the presence of the error. Are they, therefore, also in error or will correcting E alone correct them all?

When faced with the problem of ensuring the consistency of its knowledge base, two strategies are available to the learning system.

1. After correcting E, the system may test each of the concepts that depend on E. However, revising all of the concepts dependent on one that has just been modified could involve a lot of work if the network of concepts is very extensive.
2. The system may wait to see if any further errors show up. In this case, each concept will be debugged as necessary. Although more economical this method requires a method for tolerating errors if the program has been assigned a task that it must continue to perform.

It should also be noted that another source of errors in planning is noise. When a learning system is connected to a real robot, it cannot rely on the accuracy of measurements from vision systems, touch sensors, etc. Thus a plan may fail because the knowledge base does not accurately reflect the outside world. This being the case, the learning system must not revise its domain theory prematurely since there may not, in fact, be any errors. So the most prudent approach to error recovery is to delay revision of a domain theory until sufficient evidence has accumulated to suggest the appropriate changes.

Let us now give an outline of an error recovery strategy.

1. The robot learner is given a task which it is required to perform. However, its domain theory may be incomplete or incorrect.
2. In the course of performing its task, the robot's plan fails.
3. If the robot is unable to proceed by adopting another plan then it

must suspend working on its given task while it debugs its domain theory.

4. If an alternative plan is possible (for example, by reordering goals) then the new plan is attempted while storing the failed plan for future reference.
5. The robot cannot assume that the failed plan is incorrect since the cause of failure may have been due to noise. Therefore, as each plan is executed, a history of its performance is maintained; this includes the performance of the individual concepts that formed the plan.
6. The accumulation of histories is input for a DUCE style of learning system which effectively summarizes the performance of plans when it forms new concepts by generalization.
7. Since the learning program may generate alternative descriptions for the same concept, we must be able to resolve potential conflicts so that the next time a similar plan is to be created, the appropriate description is chosen.

In this final step, an assumption-based truth maintenance system (ATMS) becomes useful (de Kleer 1986*a*, *b*, and *c*). Alternative descriptions of the same concept represent different assumptions about the behaviour of the world. An ATMS provides a mechanism for carrying forward several lines of reasoning concurrently where each chain of inference is based on different assumptions. When a contradiction is encountered by one chain, it is knocked out and its assumptions invalidated. In our case, different lines of reasoning are replaced by alternative domain theories based on different concept descriptions. A failed experiment corresponds to a contradiction.

When an experiment does fail, we must not invalidate the concepts used in planning the experiment for, as mentioned earlier, the failure may be due to noise. Instead, we note the circumstances of the failure and augment the failed concept with a description of these circumstances. Several things could happen to the concept when this is done:

1. The description of the concept is modified to the extent that it becomes correct. If an alternative, correct description already existed, then the alternative domain theories of which these concepts were components, converge.
2. After several failures, there is no generalization which covers the circumstances of failure. In this case, the failures may be either attributed to noise or to some phenomenon not yet known to the system. In either case, nothing can be done.

An ATMS maintains the network of concepts that form a domain theory and stores dependencies which, when errors are found, will indicate

where other potential weaknesses in the theory lie. The ATMS also allows a learning program to experiment with alternative domain theories.

3. CAP

Hume's Concept Acquisition Program (Hume 1985, Sammut and Hume 1987) is a current example of a program working in a reactive environment. A reactive environment is one that responds to the actions of the learning program. This can be a real or simulated environment. In CAP's case we have a simulated world that contains two robots. One is under the control of the learning program which has little initial knowledge of the world. This is referred to as the *child* robot. The second robot already 'knows' about the world and can perform a variety of tasks. This is referred to as the *parent.* The child learns about the world by observing the parent performing some task and then using the observation to guide it in exploring its environment. Children often learn by trying to imitate the actions of adults. That is, when a situation arises which is similar to one where the parent has previously performed some action, the child may attempt the same action. Since the state of the world is unlikely to be identical to the initial state when the parent began its actions, imitation must also involve a degree of generalization.

To demonstrate how CAP works, we return to the example of learning the preconditions for pouring liquids from one container into another. Suppose the world consists of a solid cylinder and two cups, one with some liquid in it, the other empty. The parent robot's task is to pick up the full cup and pour the liquid contents into the empty one. At the completion of this task, the liquid is no longer in the original container, so it is not possible exactly to duplicate the same set of actions. If a child robot wishes to imitate the parent then it must be satisfied with a partial match of the starting conditions with some later state of the world. By partial match we mean that two states can be considered similar if some simple transformation can be applied to one state to turn it into the other.

Figure 2 shows the 'before' and 'after' states of the contents of cup A, being poured into cup B. Suppose the child wishes to imitate the action immediately after the parent has finished. A no longer contains the liquid; however, by comparing the descriptions of the original state and the final state we see that by substituting B for A we can use B as the source of the liquid. Similarly, substituting A for B allows A to be the destination.

Imitation based on a partial match is a useful way of learning. In this case, because A and B can be used interchangeably, the child will have learned the generalization that liquid can be poured into objects that are cups. Imitation can be viewed as an experiment whose purpose is to

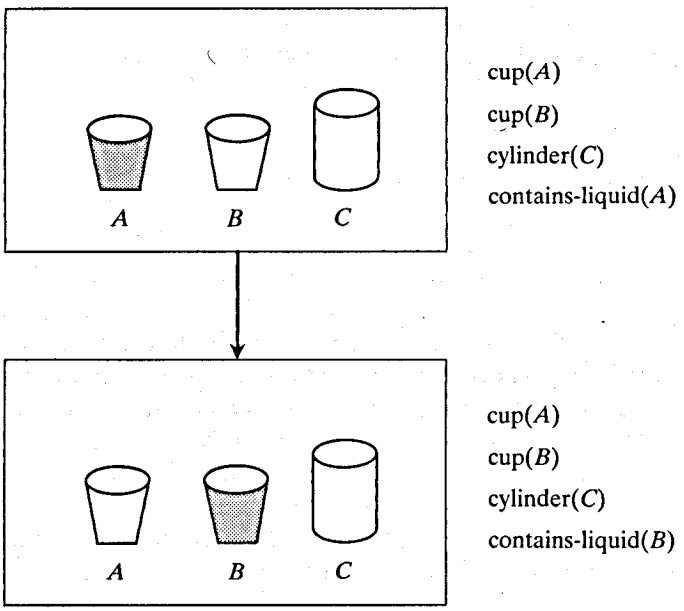

Figure 2. Finding a partial match between states.

confirm or deny a generalization. For example, after swapping A and B it can be predicted that the result will be that after pouring the liquid it remains in A. If the prediction is proved to be true then the generalization is confirmed. Let us now see how another experiment will fail to produce a predicted result but still yield useful information.

The partial match described above is obvious since one cup simply maps onto the other. However, there are more subtle similarities present in the scene. Assume that the concept:

$$\text{circular-cross-section}(X) \leftarrow \text{cup}(X).$$
$$\text{circular-cross-section}(X) \leftarrow \text{cylinder}(X).$$

is known to the system. That is, an object has a circular cross-section if it is a cup or a cylinder. This tells us that A, B, and C are all similar according to at least one criterion. Therefore, another possible substitution would allow C to be the destination of the pouring action. The previous experiment tested the effects of pouring liquid into another cup, thus permitting the generalization that any cup can contain liquid. Another experiment, this time with the cylinder, tests the generalization that objects other than cups can also contain liquids. Of course, this time the liquid does not stay in the cylinder. Thus the generalization is shown to be incorrect.

The child observes and records changes in the world as a sequence of states, where each state is represented by a description (in first order

logic) of the configuration of objects in the world. Suppose there is a sequence,

$$S_0; S_1; \ldots; S_N$$

and a current state, S. Although each S_i is a conjunction of atomic predicates, it is also useful to think of it as a set of predicates. Thus, a partial match can exist if there is some state, $S_m : 0 \leqslant m \leqslant N$ such that

$$S \cap \sigma S_m \neq \varnothing$$

That is, under some substitution σ states S and S_m share common terms in the state description. For example, if S_m consists of a full cup and an empty one and S consists of a full cup and a cylinder then a partial match exists with a substitution of the cylinder for the empty cup. However, in order for this substitution to work, it must have been recognized that cylinders and cups can be equated in some way. So before looking for a match, the system must first elaborate on the state description by using concepts, such as circular cross-section. This is done by treating the concept description as a set of forward chaining rules as described in Sammut and Banerji (1986) and is similar to the method used by Muggleton in DUCE.

The partial match permits CAP to propose the following task: since cups and cylinders are similar in at least one respect (they both have circular cross-sections) they may also be similar in their ability to contain liquids. Therefore, it should be possible to perform actions that will result in a liquid being poured into a cylinder, just as had been done with the cup (for which the cylinder has been substituted). This is the prediction, namely, that it should be possible to create a sequence of states in the world that corresponds to the sequence obtained through the matching process. The attempt to achieve the sequence in the modelled world is an experiment.

If an experiment has been concluded successfully, that is, the results match the prediction, then the child has grounds to propose a generalization. When the attempt to pour a liquid into another cup succeeds then it may be proposed that liquids can be poured into any cup. The pouring action, A_i, transforms a state S_i into another state S_{i+1}, written as

$$A_i : S_i \rightarrow S_{i+1}$$

Thus, S_i contains the preconditions for the action A_i. By generalizing S_i the applicability of the action is broadened. The next experiment, trying to pour the liquid into the cylinder, generalizes S_i even further. This generalization is incorrect; however, it can be recorded as an exception condition for action A_i. As other exceptions are encountered for A_i they may be combined with previously recorded cases. Thus, it is possible to

build up knowledge about when an action can be used and when it cannot.

Analysing why a particular substitution worked or did not work can lead to further experiments. If a particular generalization was successful, then it is worth looking for other objects covered by that generalization. If an attempt is made to broaden the generalization by substituting another object, and this fails, the difference between the object causing the failure and the previously successful objects helps to define the generalization more precisely. This refinement of the description can be achieved by trying to make the unsuccessful generalization more specific and performing another experiment.

An interesting side effect of this learning problem is that it provides a simple criterion for clustering objects into new, unnamed concepts. Objects form a cluster if they can be used in the same roles. Containers made of glass or plastic can both hold water, a brick and a table can both be used to support other objects. As an object is added to a cluster, a generalization may be performed in order to arrive at a concise description of the cluster. To perform this generalization we again refer back to DUCE's operators.

CAP's learning strategy can be summarized as follows:

1. The parent carries out a plan which results in a sequence of world states being created.
2. The sequence is compressed and stored.
3. CAP attempts a partial match between the current state and a stored state. The changes that result from the parent's actions are used as a focus for the search for a partial match.
4. Since a stored state belongs to a sequence, the nearest partial match is used to generalize the sequence starting from the matched state. The sequence thus generated attempts to predict the result of the experiment to follow.
5. A plan of action is inferred from the prediction sequence.
6. This plan is executed.
7. If the result of the experiment was as predicated then the preconditions for the actions in the plan are generalized, otherwise the exceptions conditions are generalized.
8. As long as there is nothing else to do (i.e. the parent is not doing anything that should be observed) the increasingly distant matches are used to create generalizations and experiments.

During experiments with CAP it was noted that the program could get itself into a state where further progress became impossible. If the water in all containers has been spilled then one of the basic ingredients of the

experiments is missing. To remedy this situation we 'cheat', but in an interesting way! Hume gave CAP the ability to keep more than one learning task running concurrently, so when progress is halted in one task, CAP can switch its attention to another. As well as the parent demonstrating how to pour liquid from one container to another, it also showed CAP how to create water by using a tap as a source of water. Once CAP has spilled all of its water and is unable to try pouring from one container into another, it switches its attention to learing about the use of taps. Of course, a side effect of this experimentation will be the creation of water in a container. So when the program has finished playing with taps, it can return to pouring water out of cups.

Sometimes, providing alternative tasks is not enough to prevent CAP from being blocked from further learning. Suppose, after experimenting with the tap, all containers have been filled, but the program still has not completed learning about pouring from one container to another. Even though its partial theory may tell CAP that any pouring action will result in spilled water, it may still do it if there is nothing more sensible that can be done. That is, if progress is impeded by a state that does not match anything in the program's knowledge base, CAP will perform a random and probably illegal action in order to bring about a useful new state of the world. In a sense we could say that the child throws a tantrum out of frustration, thus obeying the maxim: 'When in trouble or in doubt, run in circles, scream, and shout'!

4. RELATED WORK

Carbonell and Gil (1987) describe work on the PRODIGY general-purpose planner. Given an incomplete and possibly incorrect plan, they attempt to modify the plan when failures are encountered. An example of learning the correct plan for crafting the primary mirror of a reflecting telescope is given. (In fact, the example has the same characteristics as the classic blocks world planning problems.) In common with our earlier discussion, Carbonell and Gil point out that a plan which fails to meet expectations must contain errors. When a failure occurs, the system plans experiments to try to repair the domain theory. The theory should contain knowledge of the preconditions of each operator, the consequences, or post-conditions after applying each operator, and the objects to which it is appropriate to apply each operator. In this domain the operators include grinding, polishing, aluminizing, and cleaning objects.

The kinds of faults in the domain theory that the PRODIGY work has centred on are incomplete specification of pre- and post-conditions of operators and lack of knowledge about operator interaction. Since the domain theory is represented in first order logic, it is not surprising that

the method for planning experiments resembles Shapiro's program debugging in many ways. After a failure has forced the system to perform some debugging, the program examines the current state of the world and creates a new plan to achieve its original goal using the new domain theory. A potential hazard which the authors note is that replanning can cause a glass blank to be ground several times. If this is done too often, no glass will be left. This is similar to the problem that CAP encounters when it runs out of water during its pouring experiments.

5. CONCLUSION

We have discussed a number of problems related to incremental learning in a reactive environment. In particular, a learning system must be able to detect that it has an incorrect domain theory, and it must be able to locate the error and correct it while maintaining a consistent knowledge base. Some solutions to these problems have been suggested and an implementation of the CAP incremental learning program was described.

Acknowledgements

My thanks to Dave Hume, who designed and implemented CAP and to the members of the Turing Institute who have provided a stimulating environment in which to work during my leave from the University of New South Wales. Also to the Institute of Cybernetics of the Estonian Academy of Sciences for organizing a successful and fascinating meeting in Tallinn. My work at the Turing Institute has been supported by the Westinghouse Corporation.

REFERENCES

Carbonell, J. G. and Gil, Y. (1987). Learning by experimentation. In *Proceedings of the Fourth International Machine Learning Workshop* (ed. Pat Langley), pp. 256–66, Morgan Kaufmann, Los Altos.

Cohen, B. L. (1978). *A theory of structural concept formation and pattern recognition*, Ph.D. thesis, Department of Computer Science, University of New South Wales.

de Kleer, J. (1986*a*). An assumption based TMS, *Artificial Intelligence*, **28**, No. 1.

de Kleer, J. (1986*b*). Extending the ATMS, *Artificial Intelligence*, **28** No. 1.

de Kleer, J. (1986*c*). Problem solving with ATMS, *Artificial Intelligence*, **28** No. 1.

Hume, D. (1985). *Magrathea: A 3-D Robot World Simulation*, Honours thesis, Department of Computer Science, University of New South Wales, Sydney, Australia.

Michalski, R. S. (1983). A theory and methodology of inductive learning. In *Machine learning: An artificial intelligence approach* (eds R. S. Michalski, J. Carbonell, and T. Mitchell), pp. 83–134. Palo Alto, Tioga.

Muggleton, S. (1987). Duce, an oracle based approach to constructive induction. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Milan. See also this volume.

Muggleton, S. and Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings 5th Machine Learning Conference*. Kafumann, pp 339–52.

Quinlan, J. R. (1983). Learning efficient classification procedures and their application

to chess end games. In *Machine learning: An artificial intelligence approach* (eds R. S. Michalski, T. M. Mitchell, and J. G. Carbonell), Kaufmann, Los Altos, CA.

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

Sammut, C. A. and Banerji, R. B. (1986). Learning concepts by asking questions. In *Machine learning: An artificial intelligence approach* (Vol. 2) (eds R. S. Michalski, T. M. Mitchell, and J. G. Carbonell), Kaufmann, Los Altos, CA.

Sammut, C. A. and Hume, D. V. (1987). Observation and generalisation in a simulated robot world. In *Proceedings of the Fourth International Machine Learning Workshop* (ed. Pat Langley), pp. 267–73, Kaufmann, Los Altos, CA.

Shapiro, E. Y. (1981). *Inductive inference of theories from facts*, Technical Report 192, Yale University.

Sussman, G. J. (1973). *A computational model of skill acquisition*, Ph.D. Thesis, MIT Artificial Intelligence Laboratory.