From: AAAI-87 Proceedings. Copyright ©1987, AAAI (www.aaai.org). All rights reserved.

Explanation-Based Failure Recovery

Ajay Gupta
Hewlett-Packard Laboratories
Filton Road, Bristol BS12 6QZ, UK.
email: ag@hplb.csnet

Abstract

Interactions are inherent in design-type problem-solving tasks where only partially compiled operators are available. Failures arising from such interactions can best be recovered by explaining them in the underlying domain models. In this paper we explain how Explanation-Based Learning provides a framework for recovering in this manner. This approach also alleviates some of the problems associated with the least-commitment approach to design-type problem-solving.

I. Introduction

In the 'expert-system' literature, the need for declaratively represented 'causal' models along with compiled 'association'-based rules has been argued for reasons such as explanation, teaching and flexibility. In this paper we illustrate how compiled goal-oriented rules need to be supported by uncompiled domain principles when the problem-solver runs into failure. In this process, we also justify an observation made by Clancey [Clancey, 1983], which we believe to be true for most expert problem-solving:

> *Principles are good for summarizing arguments, and good to fall back on when you've lost grasp on the problem, but they don't drive the process of [medical] reasoning.*

Failures are particularly acute in synthesis tasks such as planning, design or control. These tasks usually require a problem-solving approach that involves *construction* of a solution in contrast to the *classification* approach which uses a pre-enumerated solution space. In these tasks, the problem-solver is given a set of goals and the operators that suggest how certain goals can be achieved or refined. These goals are hierarchically refined by applying suitable operators to generate new subgoals until the desired level of detail is reached. Most real design-type problems have some non-independent goals which lead to interactions.

Interactions can be avoided by writing operators in such a way that no unwanted relationship between the submodules is established while problem-solving. Compiling out all the interactions amounts to mapping the problem into one of classification. In general, however, it is infeasible to compile out all the interactions as they increase exponentially with the number of modules.

Another way of avoiding interactions is to anticipate them by placing constraints on the choices where incompatibility is likely to occur. By sufficiently constraining a choice point the interactions can be obviated. For instance, MOLGEN [Stefik, 1980] uses the least-commitment strategy by employing such constraint-posting. But in order to use this technique the problem-solver needs to know all conceivable interactions and the constraints to pre-empt them.

In most domains, the operators used for planning or design are only partial models of real-world actions — in particular not all postconditions of an action are known statically. This is particularly so for complex actions whose consequences will depend upon the situation in which they are employed. For instance, such operators are required in order to build plans involving simultaneous actions. Here the traditional STRIPS model of operator representation breaks down because the factors that can affect the global consequences of an operator become very large, and recording all of them will make the operator unwieldy to use. Thus the local description of operators, required for flexibility and efficiency, necessitates only partial compilation of its consequences. Interactions arise during problem-solving because of using such partially compiled operators.

Furthermore, in any real-life design task it is extremely difficult to have all requirements identified in the initial specification. Design and prototype-evaluation is a cyclical process during which the specifications are modified several times. Thus a realistic design-type problem-solver must have the capability to deal with failures as they arise during problem-solving.

Backtracking — chronological or dependency-directed — is the last resort of recovery from problem-solving failures. As we will demonstrate on some examples in the sections that follow, both of these approaches suffer from the problem of thrashing, i.e. running into identical failures repeatedly.

An alternative approach that addresses some of the above issues employs partially-compiled operators that will 'normally' produce the desired goal without interactions, but in some unusual cases when they do fail the problem-solver attempts to recover gracefully by attempting to explain it in the domain model. The technique of explain-

ing a failure is very similar in spirit to that employed in other Explanation-Based Learning (EBL) work [deJong and Mooney, 1986], [Mitchell et.al., 1986]. First we summarise the basic ideas of EBL and then illustrate using an example how it provides a useful framework for failure-recovery. Later section discuss relationships, advantages and issues for further research. This architecture has been implemented in a PROLOG-based planning system called TRAP [Gupta 1985]. The following diagram is the top-level architecture of the implemented system:

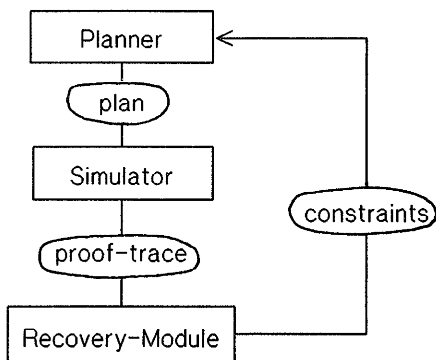

Figure 1: TRAP Architecture

II. Explanation-Based Learning

Explanation-based generalisation (EBG) is a technique for learning new concepts and rules from single examples. EBG and more generally EBL have been surveyed comprehensively in [deJong and Mooney 1986], [Mitchell et.al. 1986]. In brief, for EBG the following information is required:

Goal Concept: A definition of the concept to be learned.

Training example: A specific example.

Domain Theory: Axioms for verifying if an example has a property.

Operationality Criteria: Specifications on representation of the concepts.

EBG attempts to construct an explanation of why the training example satisfies the goal concept. This explanation takes the form of a proof tree composed of domain-theory inference rules which proves that the training example is a member of the concept. This explanation is then generalised to obtain a set of sufficient conditions for which this explanation structure holds in general. The desired general preconditions are obtained by regressing the goal concept through the explanation structure.

III. Example

Consider a robot apprentice in a metallurgy workshop where its task is to plan the production of objects with some specified properties. The planning operators, which encode metallurgical processes such as hot-rolling and heat-treatment, constitute compiled information for achieving typical goals. The apprentice also has some operators for transporting the objects around the workspace, fixing them in machines etc. An example operator for fixing an Object in a Machine is:

```
op( fix_in(Machine,Object), % name
   [gripped(Object)],        % Usewhen preconditions
   [on_side(Hand)],          % Whenuse condition
   [ ],                      % subgoals - if any
   [traverses(Object,Machine,Hand)])% postconditions
```

Furthermore, the apprentice has domain knowledge about the basic properties of materials, machines and physical process. This knowledge constitutes the domain theory and also includes certain domain requirements, e.g. tools shouldn't be damaged, and accidents should be avoided:

- melting_point(gripper,50).
- melts(X) & tool(X) → "fail".
- traverses(Object,Machine,Side) & shielded(Machine,Side) → collision(Object,Machine).

Consider a scenario where the apprentice has been given the task of making a hot-rolled bar out of a steel block. This requires heating up the block to 400 degreeC, and then fixing it on a rolling machine. Using the compiled operators the planner reduces the goal in following manner:

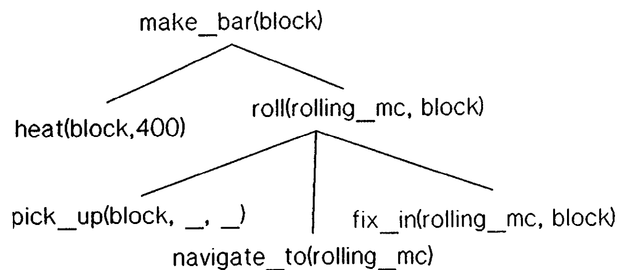

Figure 2: Hierarchical Plan

In this refinement the apprentice has the choice of holding the *Object* on its *left-* or *right-hand-side*, and also of different means for picking up an object that include using its grippers, polymer gloves or steel prongs. Because at this stage the apprentice lacks any knowledge to discriminate between the choices, it chooses to use its gripper on the right-hand-side for picking up the block leading to the following plan:

1. heat(block, 400)
2. pick_up(block,gripper,rhs)
3. navigate_to(rolling_mc)
4. fix_in(roller,block)

Before this plan is approved for 'execution' in the real world, it needs to be tested by simulation. In this process it is noticed that while picking up the heated block, the gripper will melt because its surface has a melting-point lower than the temperature of the block being picked up. This forces the plan to be rejected because it leads to a violation of the domain requirement that the tools should not damaged. Under dependency-directed backtracking, the problem-solver can revisit the choice of how to pick up the block and instead of picking it up directly with its gripper, it can use some tool. But this time it can try to pick up the hot block using polymer gloves, which will again give rise to the same failure of a melting tool. Thus backtracking does not prevent the problem-solver from making a choice that would lead to an identical failure, and hence thrash over a particular failure. Even in an assumption-based TMS [deKleer, 1986] the problem remains because it only records the instantiated nogood set.

IV. Recovery by EBG

In order to avoid this thrashing we need to note the reason behind the particular instance of failure so that the same kind of mistake is not repeated. In the above example the recovery mechanism should infer that there is a danger of damaging a tool if its melting-point is lower than the temperature of the object it is picking-up. The replacement technique that operationalises this behavior involves inferring sufficient conditions for the failure using EBG. These sufficient conditions are then rewritten into a constraint using assumptions about the tenacity of various types of choices. Negation of this condition will give the necessary conditions to avoid that failure.

In terms of EBG each instance of a failure constitutes a training example. The goal concept to be learnt is "fail". A simulation of the plan results in a proof under the domain theory. For instance, the proof tree of the failure in the example above is:

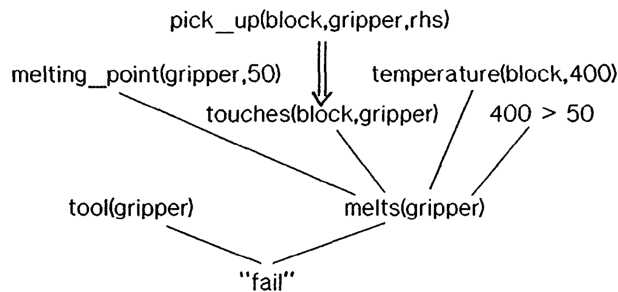

Figure 3: Proof Tree

The derivation of the above failure included the following axioms:

1. pick_up(Object, Tool, Side) ⇒ touches(Object, Tool).
2. touches(Object, Tool) & melting_point(Tool, Mp) & temperature(Object, Temp) & Temp < Mp → melts(Tool).
3. tool(Tool) & melts(Tool) → "fail".

On regressing the goal condition, we can obtain the initial premises which have to be satisfied for the proof to go through. In this process, we are identifying those properties of the objects that contributed to derivation of the failures; and it is this set of properties that must be avoided to prevent another similar failure. The above proof can be summarized as an implication of the form:

if pick_up(Object, Tool, Side) & melting_point(Tool, Mp) & temperature(Object, Temp) & Temp > Mp & tool(Tool)
then "fail".

In order that this failure can be avoided in future the planner needs to be able to test for its satisfaction whenever a choice is to be made. To make this test more efficient, the sufficient conditions for the failure need to be further transformed to get a constraint that can act as a guard at selected choice points. The constraint must obviously be evaluable at the time the choice is being considered. Additional principles used in this transformation (essentially blame-assignment) record the tenacity of different conditions. An example of such a principle would be the fact that aborting a goal (negation of goal condition) is more difficult than changing the operator to achieve it. The order in which we would be willing to give up a choice is:

1. Variable instantiation (e.g. use gripper)
2. Operator to achieve a goal (e.g. pick-up the Object)
3. Problem-solving goal (e.g. move from A to B)
4. Domain-requirements (e.g. tool should not be damaged)

This ordering is largely pragmatic. It is reasonable to expect that a problem-solver should not be allowed to change domain laws simply because some choices conflict with them. However, it is conceivable that in certain situations one might be more keen on giving up the goal rather than resatisfying it (e.g. in time-critical planning), or that certain domain requirements can be waived to meet a critical goal. In our implementation these choices, which in the framework of EBG form the operationality criteria, have been hardwired so that goal regression stops at the operators. But in general a flexible control strategy can be used to dynamically compute this operationality criteria [deJong & Mooney, 1986].

In the above example, after applying the simplification suggested above, we get the following constraint that records the necessary conditions for avoiding the failure:

action: pick_up(Object, Tool, Side)

constraint: if melting_point(Tool, Mp) & temperature(Object, Temp) then not(Mp < Temp).

This constraint is added as another field in the operator definition and used as a guard over the choices to prevent all the instantiations of this derivation of the failure.

V. Compilation

In the above example the derived constraint did not depend upon any condition other than those that were present in the state in which the failure occurred. In general a failure may depend upon choices that have been made by past actions. Continuing in the above plan, the action *pick up(block, steel prongs, rhs)* is to be followed by the action *fix in(rolling mc, block)*. If the apprentice knows that *shielded(rolling_mc,rhs)* then the plan would fail because of *collision(rolling_mc,block)* which is recognised as a violation of a domain requirement.

1. pick_up(Object,Tool,Side) ⇒ on_hand(Object,Side).
2. fix_in(Machine,Object) & on_hand(Object,Side) → traverses(Object,Machine,Side).
3. shielded(rolling_mc,rhs).
4. traverses(Object,Machine,Side) & shielded(Machine,Side) → collision(Machine,Object).
5. collision(Machine,Object) & machine(Machine) → "fail".

In this case the derived constraint would be placed on the action *pick_up()*, because that is where the only relevant variable instantiation choice was made, but it depends upon the action *fix_in()* that comes later in the plan. The scenario is illustrated in figure 4.

In general such a constraint will not be evaluable at the action *pick_up* because there is no a priori knowledge that it would be followed by the action *fix_in* in a particular problem. Further, the action *pick_up* need not be immediately followed by the action *fix_in* for this failure to occur. So long as the culprit condition, in this case *on_side(block,rhs)*, is not disturbed by intermediate actions the same failure can be derived (essentially a consequence of the frame assumption).

In our current implementation we have taken the restricted generalisation approach [Mitchell et.al., 1986]. We compile, in the form of an abstract operator, the subtree of the goal-refinement tree (generated while planning) that would include all the actions used in the proof of the failure. In the above example, from the goal-refinement tree shown, it is clear that the subtree subtended at *roll()* covers all the actions participating in the above failure condition. A new operator *roll'()* is created with its subgoals as the actions on the frontier of the subtree. As this new operator ensures that the sufficient condition for the failure would hold, we attach the derived constraint to this new operator. As has been recognised in [deJong & Mooney, 1986], this approach leads to under-generalisation because now the constraint would be applicable only if a specific sequence of actions are executed. In general, not only is it difficult to describe the above predicate, that represents if a condition is carried from one action to another action unchanged by intermediate actions, it is also impossible to evaluate such a condition while planning.

In the process of generating these constraints we are generalising from a failure instance. Compilation of these constraints into operators results in specialisation of the operators. Increasing compilation leads to increased efficiency, but in our implementation the general versions of the operators are retained for the purposes of flexibility.

VI. Application Criteria

For explanation-based failure recovery to be useful the generated sufficient conditions for failure need to be more general than the actual sequence which led to the failure: otherwise the technique degenerates into dependency-directed backtracking. Non-trivial constraints can be inferred only if the domain model has been represented intensionally. For instance, if the problem-solver stored only the fact that touching the hot block with the gripper damages it, without recording underlying reasons, it can only infer that the gripper should not be used for picking hot blocks. Backtracking makes the assumption that alternatives at a choice-point are 'independent', i.e. they have no relation with one another, which need not be the case. We have noted that by using a domain model — sufficiently deep that these relationships can be inferred, a more general condition than that encoded in the nogood sets can be rejected after failure.

One of the outstanding problems with EBG, as pointed out in [Mitchell et.al., 1986], is generating an explanation in an incomplete or undecidable domain model. In the discussion above the simulation needs to be exhaustive enough to detect every failure derivable under the domain theory. It is clear that complete simulation of a sce-

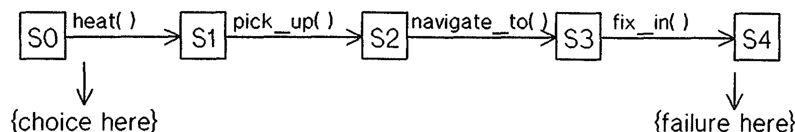

Figure 4: Complex Failure

nario is computationally prohibitive. We need some mechanisms for guiding the search for the failures. In practice there are two approaches:

a. simulation can be localised to look for certain kinds of problems that are more likely to occur.

b. in situations where it is safe, the plan can be executed in the real-world and its results explained if they turn out to be different to expectation.

In the context of EBL, [deJong & Mooney, 1986] have noted the first alternative as learning under external guidance, and the second as learning by observation. In terms of plan-time failure-recovery the techniques of partial simulation and execution monitoring are being actively pursued. But how a proof is arrived at is independent of the techniques used in the recovery module.

The generalisation procedure currently used in TRAP is similar that in [deJong & Mooney, 1986]. The simulator generates the *SPECIFIC* instance of the proof-tree, and records the axioms that were used in the process. Using this proof-tree and the recorded dependencies, the recovery mechanism generates the *GENERAL* version of the explanation structure.

VII. Relationships

In MOLGEN [Stefik, 1980], where the least-commitment strategy is used for object-selection, backtracking has been replaced by the forward-reasoning involved in constraint-propagation. It is well known that this forward reasoning can be equally computationally expensive if there are a sufficient number of applicable but irrelevant constraints in a situation. The problem is identification of relevant constraints. Risk-and-recover contrasts with the very cautious approach of least-commitment which saves the cost of failure-recovery but trades it off for the cost of making sure that a step is right. We have suggested how the constraints can be generated dynamically to cover failures as they are encountered. In fact, if in the absence of some constraints the plan fails due to interactions, the resulting failure can be used to infer the constraints which would avoid it in future. In this manner, the problem-solver becomes more careful after encountering a failure, and avoids being an 'pessimist' that tests for everything that can go wrong before taking a step.

Opportunistic planning, suggested as a model for human planning [Hayes-Roth, 1983], emphasises the need of data-directed reasoning along with goal-directed reasoning. The architecture presented in this paper is a restricted interpretation of the opportunistic model where detailed refinements can suggest modifications to previously taken abstract decisions, but only when the problem-solver runs into failures. By thus restricting the 'data-directed' guidance to be invoked only when the goal-directed approach does not quite work, we can get a computationally realistic control strategy. The idea of using sufficient conditions for a failure has been proposed as the avoidance method in [Hayes-Roth, 1983].

VIII. Conclusions

We have presented a framework based on explanation-based learning that illustrates the role of a domain-model in supporting the compiled goal-oriented operators in the design-type problem-solving tasks. This support is required in an evolving system, because at any point in time, although the domain model can be expected to be complete, compilation of the goal-oriented operators would be necessarily incomplete. This framework alleviates the three difficulties mentioned in the introduction: compiling out all the interactions, identifying applicable constraints in order to use least-commitment and the thrashing inherent in backtracking.

We are currently investigating further into reasoning about temporally ordered actions in generalised failure constraints and integrating this technique with other recovery techniques such as goal-reordering [Tate, 1977] for dealing with failures due to interacting sub-goals.

Acknowledgements

John Lumley and Stefek Zaba provided extensive comments on the presentation of this paper. The Author was financially supported by Inlaks Foundation Scholarship while at the University of Edinburgh.

References

[Clancey, 1983] W.J. Clancey. An Epistemology of a Rule-Based Expert-Systems: A Framework for Explanation. *Artificial Intelligence*, 20(3):215–251, (1983).

[deJong and Mooney, 1986] G. deJong and R. Mooney. Explanation-Based Learning: An alternative view *Machine Learning*, 1(2):145–176, 1986.

[deKleer, 1986] J. deKleer An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, (1986).

[Gupta, 1985] Ajay Gupta. *Failure Recovery Using a Domain Model.* MPhil Thesis, Dept. of AI, University of Edinburgh 1985.

[Hayes-Roth, 1983] F. Hayes-Roth. *Using Proofs & Refutation to learn from Experience.* In *Machine Learning: an Artificial Intelligence Approach*, Michalski, R.S., et. al. (eds.), Tioga, Palo Alto, CA 1983.

[Mitchell *et. al.*, 1986] T. M. Mitchell, et. al. Explanation-Based Generalisation: A Unifying View. *Machine Leaning*, 1(1):47–80, 1986.

[Stefik, 1980] Mark Stefik. *Planning with Constraints.* PhD Thesis, Dept. of Computer Science, Stanford University 1980.

[Tate, 1977] Austin Tate. Generating Project Networks. In *Proceedings IJCAI-77*, pp 888–893, MIT, Cambridge, International Joint Conference on Artificial Intelligence, 1977.