Integrating Failure Recovery with Planner Debugging*

Adele E. Howe
Computer Science Department
Colorado State University
Fort Collins, CO 80523
howe@cs.colostate.edu

Abstract

Automated failure recovery and debugging are two common methods of reducing the impact of failures. This paper proposes an integration of the two methods in a system for helping designers tune failure recovery and debug a plan knowledge base. The system will collect information about failures and their recovery during planner execution and use that information to identify possible flaws in failure recovery and in the plans. The system will be an automated and extended version of previous research on Failure Recovery Analysis [Howe, 1993].

Introduction

Consider the following future scenario: you install a delivery robot on your manufacturing floor. You watch it carefully for the first few days without incident. Then one day, you return to discover that it emptied the delicate, special-order parts five feet from their destination – why? Assuming our robot remembers everything, you can examine a trace of its actions and its sensory inputs, but likely the trace is quite large and superficially looks like any other day. Instead, you could run an analysis program that will search the trace for significant patterns of actions and failure recovery leading to this failure and others, identify long-term temporal relations, generalize those patterns over the behavior of the past week and so identify the cause of the unexpected need for a part reorder. This paper proposes how this future scenario might be realized by integrating failure recovery and debugging.

In general, two approaches have been adopted to improving planner reliability: automated failure recovery (e.g., [Gini, 1988, Simmons, 1988, Wilkins, 1985]) and debugging (e.g., [Sussman, 1973, Hammond, 1986]). The first is designed to fix failures on-line that are too difficult or costly to avoid. The second eliminates or ameliorates the effects of failures caused by the actions of the planner itself. For complex planners, neither approach is sufficient by itself. We could pursue the two approaches separately: debug as much as possible and then build a good failure recovery component to handle what remains. In which case, we place considerable trust in the thoroughness of the two phases. Alternatively, we can integrate the two approaches.

Exploiting Failure Recovery

Integrating failure recovery and debugging provides several advantages over maintaining their separation. First, we need not interrupt the on-going process (planning and execution) for debugging. Second, we avoid the need to replicate the conditions that led to a particular failure. Because many planner failures can be pernicious, caused by subtle interactions within or across plans, they can be extremely difficult to replicate.

Third, the combination allows us to exploit information gathered during failure recovery as additional evidence of causes of failure. Failure recovery uses plans in ways not explicitly foreseen by the planner's designers, but not forbidden or prevented by them either. Failure recovery may repair plans by adding or replacing portions of them. As a result, the plan may include plan fragments that are juxtaposed in orders and context not envisioned by the designers. Additionally, failure recovery itself may influence which failures occur; fixes to failures may cause other failures later on.

Finally, information gathered as part of debugging can be used to improve the responses of automated failure recovery. Failure recovery influences which failures occur; minor changes in the design of failure recovery produce significant changes in the number and types of failures [Howe and Cohen, 1991]. Failure recovery may cause other failures downstream or may simply not be as efficient as possible. In sum, the two processes, failure recovery and debugging, can support each other.

*The original research, as part of the author's PhD, was supported by a DARPA-AFOSR contract F49620-89-C-0011. This research is currently supported by National Science Foundation Research Initiation Award #RIA IRI-9308573.

An Integrated Methodology for Improved Reliability

One way to integrate failure recovery and debugging is to use information obtained during failure recovery as input for assisted debugging. Figure 1 shows how the two parts, failure recovery and *Failure Recovery Analysis* (a partially automated debugging procedure), are integrated. Failure recovery is like a loop within the debugging loop. Failure recovery repairs failures that arise during normal planning and acting. Failure Recovery Analysis (FRA) "watches" failure recovery for clues to bugs in the planner and informs the designer of possible bugs. FRA statistically analyzes execution data for significant patterns of failure and then employs a weak model of planner and environment interactions (stereotypical explanations) to hypothesize how the observed failure patterns might have been produced. Because FRA uses a weak model, it can localize a variety of bugs and should be appropriate for many planners, but it cannot guarantee that it will find all bugs or correctly explain all the failures. It is most appropriate for finding occasionally repeating bugs that are due to common detrimental planner interactions.

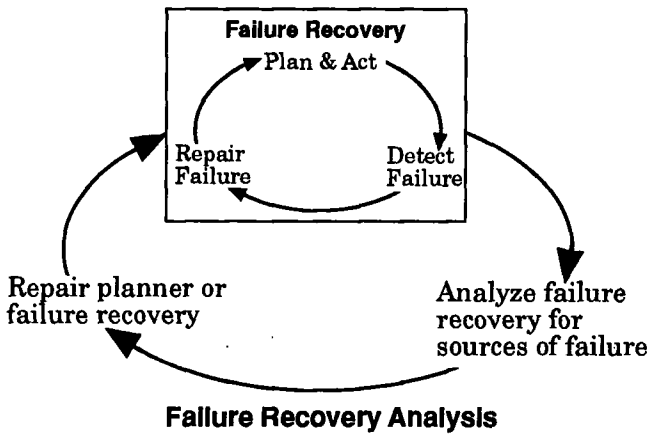

Figure 1: Relationship of failure recovery to debugging (Failure Recovery Analysis)

This approach increases the reliability of the planner by tuning failure recovery through debugging and identifying potential flaws in the planner's knowledge base. The approach was introduced in my thesis research [Howe, 1993] and is currently being extended and automated.

Tuning Failure Recovery

To tune failure recovery, we begin with a simple control strategy, a flexible method selection mechanism and a core set of recovery methods. We test the performance of that set, use the performance to motivate the method selection and then refine the set by evaluating subsequent failure recovery performance in the host environment.

Failure recovery is initiated in response to detecting a failure. Failure recovery iteratively tries recovery methods until one works, at which point the plan is resumed. The recovery methods make mostly simple repairs to the structure of the failed plan. Consequently, these methods can be used in different situations, do not require expensive explanation, and offer a response to any failure. This strategy sacrifices efficiency for generality and results in a planner capable of responding to most failures, but perhaps in a less than optimal manner. Additionally, it exploits the contents of the plan knowledge base, and when information is recorded about the knowledge base fragments used, these methods indirectly test for flaws or inadequacies in the knowledge base.

Starting from the core set, one can enhance recovery by gradually broadening the classification of failures and systematically adding new recovery methods to address them, evaluating the recovery methods as they were added. Recovery methods are evaluated in two ways: comparing performance in terms of an expected cost model and checking whether recovery detrimentally interacts with portions of the plan in progress. The expected cost model summarizes the combined cost of unsuccessful and successful recovery attempts and provides a means of comparing the contribution of new methods. The other form of evaluation, checking for detrimental interaction, is described in the next section.

Debugging the Planner and Failure Recovery

The purpose of FRA is to identify cases in which plans may influence, exacerbate or cause failure. In FRA, plan debugging proceeds as an iterative redesign process in which the designer analyzes execution traces, locates flaws, and modifies the planner to remove the flaws. (Figure 2 depicts a fully automated version of this process.) The process continues until the designer is satisfied that enough flaws have been removed. Currently, FRA is a partially automated procedure which is controlled by a human designer, who participates at every step in the process.

First, the designer starts by running the planner in its environment and collecting execution traces of what failures occurred and how they were repaired. Thus, failure recovery provides the input for the FRA process. Second, the execution traces are searched (by a program) for statistically significant co-occurrences (called *dependencies*) between recovery efforts and subsequent failures [Howe and Cohen, 1993]. Dependencies tell the designer how the recovery actions influence the failures that occur and how one failure influences another. Dependency detection involves counting the number of times particular repairs and failures co-occurred (meaning one preceded the other) and statistically comparing

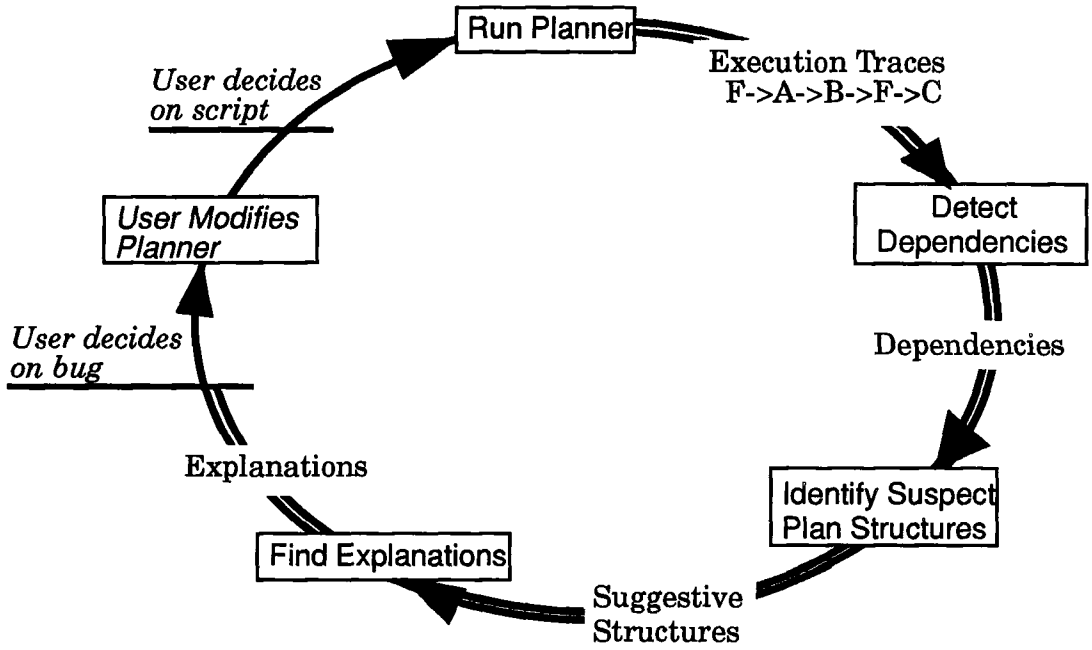

Figure 2: Enhanced debugging cycle for FRA

that count to the rest of the execution data to determine whether the co-occurrence was significant.

Third, the designer selects one of the dependencies for further attention and tries to determine how the planner's actions might have related to the observed dependency. The dependencies are mapped to actions in plans and then the plans are searched for structures that involve the actions and that are known to be susceptible to failure. These structures are called *suggestive structures* because they are suggestive of possible bugs. Suggestive structures can be combined to form different explanations of failures and indicate different repairs. Fourth, the designer matches the suggestive structures for the dependency to a set of possible explanations and modifications.

At the end of the cycle, the designer chooses the most likely explanation based on her or his understanding of the planner and modifies the planner to remove the suspected flaw. The nature of the revision is left to the designer. In effect, the selected explanation and modification constitute a hypothesis about what caused the failure. Repeating the cycle tests the hypothesis.

The cycle begins again with the designer running the planner. In the next time around, the designer can search for more flaws to fix and can determine whether the modification achieved the desired effect: the observed dependency disappeared and the incidence of failure changed for the better.

Current Status

The described integration of failure recovery and debugging has been explored in a set of four experiments with the Phoenix planner. Over the course of the experiments, failure recovery has been iteratively improved by tuning the selection strategy, adding new recovery methods, and repairing bugs in the plan knowledge base. The result is that the incidence of failure has been gradually reduced from .42 to .33 failures per hour.

At present, the integration of failure recovery and FRA is limited in several ways. First, the designer currently participates at every step in the FRA process, designing the experiments, deciding where to focus attention, judging whether the explanations for failures are correct and ultimately determining how to fix the planner to repair the bug. Second, the analysis is limited to execution traces which include failures and recovery actions only, and constrained patterns of a failure and the failure and recovery that immediately preceded it. Thus, the analysis cannot detect temporally separated cause and effect or arbitrary patterns of cause and effect. Third, the cases have been limited to the Phoenix planner and its environment. Because it includes an experiment interface and simulates a challenging environment that would be difficult to model completely, the Phoenix system has facilitated this research; however, the techniques need to be generalized and extended to other domains.

To address these limitations, we need to develop integrated tools to automate the tedious parts of the process and to support the difficult task of interpreting the

results of the experiments. Current research focuses on additional automation and generalization beyond the current constrained patterns and single planner.

At present, the experiment design for evaluating failure recovery and the analysis procedure FRA form a collection of tools and heuristics loosely organized into a semi-automated procedure for evaluating failure recovery performance and identifying bugs in Phoenix plan knowledge. The first step in building a debugging tool is to tightly integrate the existing functions and automate portions of the remainder of the procedure. The system will support both unfocused exploration and focused debugging. Unfocused exploration includes running benchmarks to ensure that the system maintains a particular level of performance and analyzing execution data for evidence of previously unsuspected bugs. Unfocussed debugging will work as follows. Using scripts, FRA will direct the entire process from gathering execution information up to the modification of the planner, presenting the designer with a summary of the execution analysis (dependency detection) and hypotheses about bugs in the planner. The designer then could choose whether to modify the planner or to conduct focused debugging to gather evidence in support of the hypotheses. Figure 2 shows this version of the FRA process. The designer will intervene only to select or create a script, review the results of the execution traces and analysis, and modify the planner; the rest of the procedure will be automated.

Focused debugging involves testing the system for the presence of particular bugs and evaluating the performance of specific design changes. Scripts will be included for repeating the previous experiment setup (with the addition of what is being tested) and comparing the results of the new execution to the previous execution. Facilities for defining new scripts will be included for designers who wish to test more during focussed debugging.

Additionally, we are generalizing the process by enhancing the underlying statistical analyses to support more patterns and looking for additional planners and environments. One of the most promising approaches for enhancing the statistical analysis is to apply local search to find the most significant dependencies of much longer, more complicated patterns. This approach has two advantages over the current approach: it finds partial order patterns (e.g., A comes sometime before B with other failures in between which comes sometime before C) and it finds only the most significant. Thus, the patterns are more general, and the user is not deluged with dependencies.

We are looking for a planner and simulated environment in which to test the process and gather more knowledge about failures. The most promising would be one that shares some characteristics with Phoenix, but presents new challenges as well. For example, the planner might be resource constrained but with different resources than Phoenix. Planners in resource-constrained environments, like Phoenix, often fail due to resource bottlenecks; some agent, process or part of the plan consumes too many resources or does not perform up to required levels, resulting in a bottleneck that produces failures in other processes or parts of the plan. Detecting a bottleneck may require examining data on the allocation and use of resources or the rate of processing. Such features are not currently included in FRA, but should be.

References

[Gini, 1988] Maria Gini. Automatic error detection and recovery. Computer Science Dept. 88-48, University of Minnesota, Minneapolis, MN, June 1988.

[Hammond, 1986] Kristian John Hammond. *Case-Based Planning: An Integrated Theory of Planning, Learning and Memory.* PhD thesis, Dept. of Computer Science, Yale University, New Haven, CT, October 1986.

[Howe and Cohen, 1991] Adele E. Howe and Paul R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 801–808, Anaheim, CA, July 1991.

[Howe and Cohen, 1993] Adele E. Howe and Paul R. Cohen. Detecting and explaining dependencies. In *Working Notes of the Fourth International Workshop on AI and Statistics*, January 1993.

[Howe, 1993] Adele E. Howe. *Accepting the Inevitable: The Role of Failure Recovery in the Design of Planners.* PhD thesis, University of Massachusetts, Department of Computer Science, Amherst, MA, February 1993.

[Simmons, 1988] Reid G. Simmons. A theory of debugging plans and interpretations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 94–99, Minneapolis, Minnesota, 1988. American Association for Artificial Intelligence.

[Sussman, 1973] Gerald A. Sussman. A computational model of skill acquisition. Technical Report Memo no. AI-TR-297, MIT AI Lab, 1973.

[Wilkins, 1985] David E. Wilkins. Recovering from execution errors in SIPE. Technical Report 346, Artificial Intelligence Center, Computer Science and Technology Center, SRI International, 1985.