A Theory of Debugging Plans and Interpretations

Reid G. Simmons
MIT Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139
REID@OZ.AI.MIT.EDU

Abstract

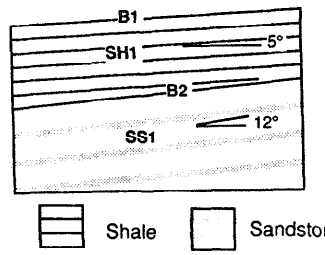
We present a theory of debugging applicable for planning and interpretation problems. The debugger analyzes causal explanations for why a bug arises to locate the underlying assumptions upon which the bug depends. A bug is repaired by replacing assumptions, using a small set of domain-independent debugging strategies that reason about the causal explanations and domain models that encode the effects of events. Our analysis of the planning and interpretation tasks indicates that only a small set of assumptions and associated repair strategies are needed to handle a wide range of bugs over a large class of domains. Our debugging approach extends previous work in both debugging and domain-independent planning. The approach, however, is computationally expensive and so is used in the context of the Generate, Test and Debug paradigm, in which the debugger is used only if the heuristic generator produces an incorrect hypothesis.

1 Introduction

Employing heuristic rules to generate an initial hypothesis and then debugging if the hypothesis is incorrect has proven to be a useful problem solving strategy (e.g., [Marcus], [Hammond], [Sussman], [Simmons]). The efficacy of this strategy depends on the presumptions that, for most problems, the heuristics can be used to efficiently generate hypotheses that are correct or nearly so and that debugging hypotheses, while not necessarily efficient, is robust enough to solve the problems handled incorrectly by the heuristics.

We present a theory of debugging applicable for planning and interpretation problems. The theory is robust, handling a wide range of bugs that arise in a large variety of domains. Debugging is accomplished using four general reasoning techniques: 1) assumptions underlying bugs are located by tracing through causal dependency structures that explain why bugs arise; 2) the directions in which to change assumptions are indicated by regressing values back through the dependencies; 3) bugs are repaired by using domain-independent repair strategies that replace faulty assumptions; and 4) the goodness of proposed repairs is estimated by determining its effect on the overall problem — whether it introduces new bugs or serendipitously repairs other existing bugs.

*Current address: Computer Science Department, CMU, Pittsburgh, PA.



1. Deposit sandstone, creating **SS1**
2. Tilt by 7°
3. Deposit shale, creating **SH1**
4. Tilt by 5°

A. The Goal State B. One Plausible Solution

Figure 1: A Geologic Interpretation Problem.

We are exploring these ideas within the Generate, Test and Debug problem solving paradigm [Simmons]. The GTD paradigm was developed for interpretation and planning tasks, both of which are of the form "given initial and goal states, find a sequence of events that could transform the initial state into the goal state" GORDIUS, our implementation of GTD, has been used to solve problems in several domains, including our primary domain of geologic interpretation, blocks-world planning, and the Tower of Hanoi problem.

In geologic interpretation, the task is to find a sequence of events that plausibly explains how a vertical cross-section of a geologic region (the goal state) was formed starting from the initial state of bedrock under sea-level (see Figure 1). The goal state describes the compositional, topological and geometric aspects of the region. For example, the goals in Figure 1a are to explain why formations **SH1** and **SS1** are composed of shale and sandstone, respectively; why **SH1** is over **SS1**; why **SH1** and boundary **B1** are oriented at 5°; and why **SS1** and **B2** are oriented at 12°.

In GTD, the generator constructs an initial hypothesis by matching a library of heuristic (associational) rules against the initial and goal states and by composing the partial sets of events suggested by each rule. The initial hypothesis is then tested. If the test succeeds, the hypothesis is accepted as a solution. If it fails, the tester passes to the debugger causal explanations for the bugs detected.

The debugger uses the four reasoning techniques enumerated above to locate and replace faulty assumptions underlying the bug. When the debugger estimates that all bugs have been repaired, the modified hypothesis is submitted to the tester for verification. This debug/test loop continues until the test succeeds. Alternatively, if the debugger appears to be moving far from a solution, the generator may be invoked to produce a new hypothesis.

1. Deposition1 of sandstone, creating **SS1**
2. Tilt1 of 12°
3. Deposition2 of shale, creating **SH1**
4. Tilt2 of 5°

Figure 2: Initial, Buggy Interpretation of Figure 1a.

For the problem in Figure 1a, the generator interprets that the deposition of **SH1** occurs after the deposition of **SS1**, using the heuristic that an overlaying sedimentary formation is younger since deposition occurs from above. To interpret the orientation of **SS1** and **B2**, the generator uses the heuristic that a sedimentary formation oriented at a non-zero angle θ was formed by deposition followed by a tilt of θ. This rule derives from the fact that, in our model of geology, deposition occurs horizontally and tilt acts to change orientations. Another application of this rule is used to interpret the orientations of **SH1** and **B1**. Combining all the constraints (and linearizing the events), the generator produces the initial hypothesis in Figure 2.

In testing this hypothesis, two bugs are detected — in Figure 1 the orientation of both **SS1** and **B2** is 12°, while the orientation predicted by simulating the hypothesis of Figure 2 is 17°. Causal explanations for why the bugs arise are passed to the debugger. For example, the orientation of **SS1** is not 12° because it was zero when deposited, **Tilt1** incremented it by 12°, and then **Tilt2** incremented it by an additional 5°. The debugger analyzes whether replacing any of the assumptions underlying the bug will repair it. Several modifications to the hypothesis are proposed, including replacing the assumption that the parameter value of **Tilt1** is 12° with the assumption that the value is 7°, producing the solution of Figure 1b.

Our theory of debugging and general debugging algorithms are presented in Section 2, while Section 3 illustrates the debugging of the hypothesis in Figure 2 in more detail. In Section 4, we analyze the completeness, coverage and efficiency of our theory of debugging. Section 5 presents a comparison with other debuggers and with domain-independent planners.

2 A Theory of Debugging

Our theory of debugging is based on the simple observation that the manifestation of a bug is only a surface indication of some deeper failure. In particular, bugs ultimately depend on the assumptions made during the construction and testing of hypotheses.

A bug, for our purposes, is an inconsistency between the desired value of some expression and its value as predicted by the tester. If the predicted value does not match the desired value, it must be that one (or more) of the underlying assumptions is faulty and needs. The debugger proposes changes that either make the predicted value match the desired value, or make the desired value no longer needed to solve the problem.

The debugger uses several reasoning techniques to locate and replace assumptions underlying bugs. The debugger locates the assumptions underlying a bug by analyzing causal dependency structures, which are acyclic graphs that represent justifications for the predicted (and desired) states of the world. The dependency structures capture an intuitive notion of causality in which time, persistence, and the effects of events are represented explicitly. In GORDIUS, the dependencies are produced as a by-product of the tester's causal simulation algorithm and are represented and maintained using a TMS [McAllester].

Each bug actually has two dependency structures — one explains how events cause the predicted value to arise; the other is an explanation for why the desired value is needed (the latter often consists of only a single assumption). For example, Figure 3 illustrates the dependency structures for why the orientation of **SS1** is predicted to be 17°, while it is desired to be 12°, as measured in Figure 1a.

In Figure 3, **SS1.orientation@Plan-end** refers to the orientation of the **SS1** formation at **Plan-end**, the time associated with the goal diagram in Figure 1a. **Persistence(SS1.orientation, Tilt2.end, Plan-end)** means that the orientation of **SS1** did not change from the end of the **Tilt2** event through **Plan-end**. The statement **Change(+, SS1.orientation, 5°, Tilt2)** means that during **Tilt2** the orientation of **SS1** increased by 5°. The dependency structure indicates that this change happens because the **Tilt2** event is predicted to occur, the parameter value of **Tilt2** is 5°, and the **SS1** formation exists at the time **Tilt2** occurs.

The assumptions underlying a bug are located by tracing back through the dependency structures to their leaves (the boxed statements in Figure 3). To indicate the direction in which to change underlying assumptions, the debugger regresses values and/or symbolic constraints back through the dependency structures. For example, regressing 12°, the desired value of **SS1.orientation@Plan-end**, through the dependencies of Figure 3 indicates that **SS1.orientation@Tilt2.end + Theta2** should also be 12°. Symbolically solving for **SS1.orientation@Tilt2.end** indicates that it should be **12 − Theta2**, which the debugger simplifies to 7°, since **Theta2** is known to be 5°.

Regressing further indicates that the desired value of **SS1.orientation@Tilt1.end + Theta1** is 7°; solving yields:
Theta1 = 7° − SS1.orientation@Tilt1.end. Since the predicted orientation of **SS1.orientation@Tilt1.end** is zero, the regression indicates that the bug can be repaired by changing **Theta1** to 7°.

The search for underlying assumptions is pruned if the regression indicates that some expression cannot be changed in any way to repair the bug. For example, if we desire **SS1.orientation@Plan-end** to be greater than **SS1.orientation@Tilt2.start**, regression yields the constraint **SS1.orientation@Tilt2.start + Theta2 > SS1.orientation@Tilt2.start**, which is simplified to **Theta2 > 0**. Since this constraint does not mention **SS1.orientation@Tilt2.start**, implying there is no way to change its value to repair the bug, the debugger does not try to locate its underlying assumptions.

Bugs are repaired by using domain-independent repair strategies that reason about the dependency structures, the regressed values, and causal domain models that explicitly detail the preconditions and effects of events. For example, if a bug depends on an assumption about the value of an event's parameter, the repair strategy is to

change the parameter value to its regressed value.

We have developed repair strategies for six types of assumptions that account for most of the bugs that appear in the domains we explored:

1. **Occurs(type, event)** is the assumption that an **event** of **type** occurs from **event1.start** to **event1.end**. One debugging strategy associated with this type of assumption is to delete the event altogether. This strategy is applicable if removing the event will achieve the desired value of the bug. Another strategy is to replace the event with a similar event, where "similar" means that the event has the same desirable effects (i.e., achieves the same goals) as the original event but avoids the bug. If the bug stems from the fact that not all of the **event**'s preconditions hold, the strategy is to find another event that achieves the same goals but does not have the offending preconditions. If some effect of the **event** helps to cause the bug, the strategy is to find an event without the offending effect that still achieves all the goals. For example, the debugger tries to repair the bug in Figure 3 by looking for an event similar to **Deposition1** that can create a sandstone formation oriented at -5°, rather than horizontally.

2. **Parameter-of(event, formal, actual)** is the assumption that the **formal** parameter of **event** is bound to the **actual** value. The repair strategy is to change the value of the parameter, where the new parameter value is indicated by the regression. For example, as described above, regression through Figure 3 indicates that changing **Theta1**, the parameter of **Tilt1**, from 12° to 7° will repair the bug.

3. **event1.end** < **event2.start** is the assumption that **event1** precedes **event2**. In our models, temporal-ordering assumptions typically support assertions that the attribute of some object persists in value from **event1.end** to **event2.start**. The debugger reorders the two events if the regressed value of the attribute is achieved at the start of **event1**. In Figure 3, for instance, **Tilt1.end** < **Tilt2.start** cannot be reordered to repair the bug since at the start of **Tilt1** the orientation of **SS1** does not equal 7°, the value desired at **Tilt2.start**.

4. **CWA(attribute.object, t1, t2)** is a closed-world assumption that no known event affects the attribute (it persists) between times **t1** and **t2**. The repair strategy is to insert an event occurring between **t1** and **t2** that can affect the attribute in such a way as to achieve the desired (regressed) value of the attribute at time **t2**. For example, the debugger proposes replacing the assumption **CWA(SS1.orientation, Tilt2.end, Plan-end)** with assumptions that a new tilt event with tilt parameter -5° occurs, where the new event is constrained to occur between **Tilt2.end** and **Plan-end**. The parameter value of -5° is determined analogously to that in #2 above. The repair strategy also determines whether existing events can affect the attribute to achieve its desired value. If so, temporal orderings are changed to make the event fall within the persistence interval (between **t1** and **t2**).

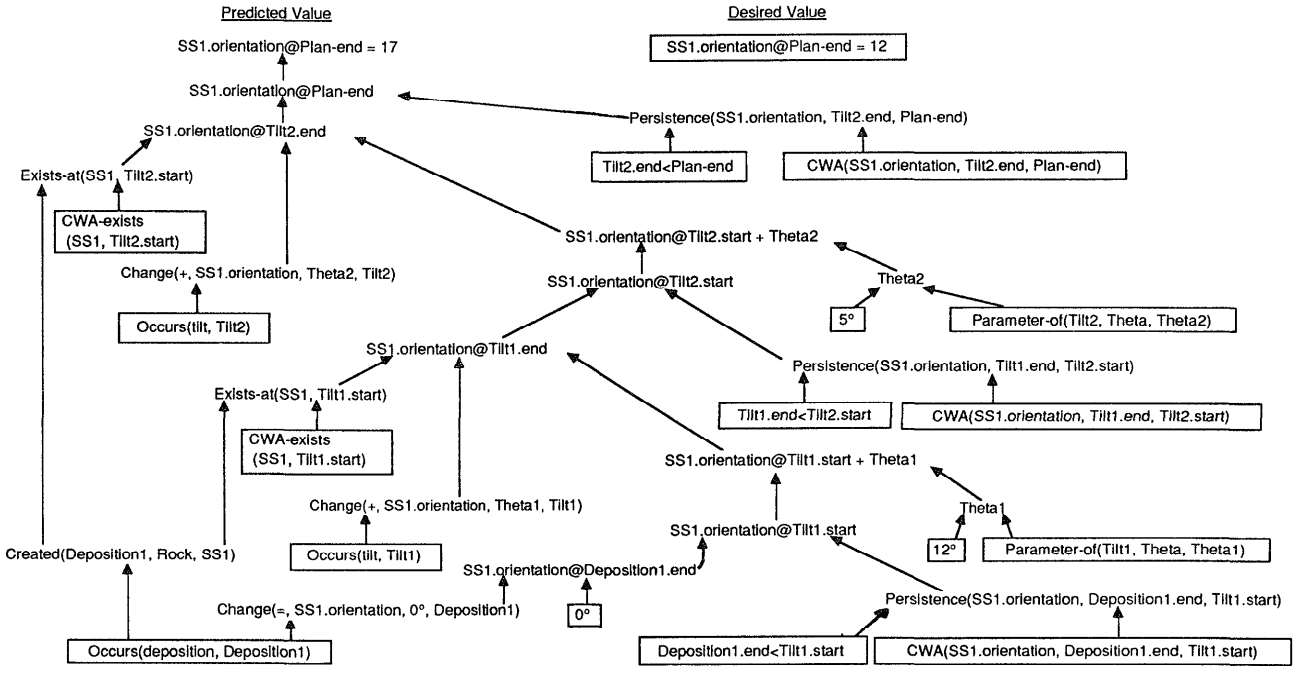


Figure 3: Causal Dependency Structure for Bug that the Orientation of **SS1** at time **Plan-end** is not 12°.

5. **CWA-Exists(object, t1)** is a similar closed-world assumption indicating that the **object** continues to exist at time **t1** since no event is known to have destroyed it. The repair strategy is to insert an event before **t1** that can destroy the **object**. For example, one way to repair the bug of Figure 3 is to prevent **Tilt2** from affecting **SS1**. Since tilting applies only to formations that exist at the time of the tilt, we can accomplish this by assuming that some erosion event totally erodes away **SS1** before **Tilt2.start**. Overall, however, this is a rather poor repair since it ends up destroying the complete geologic region.
6. **CWA-Object(type, Object1, ..., Objectn)** indicates that **Object1-Objectn** are the only known objects of **type**. The assumption is used in reasoning about quantified goals — a goal of the form **(forall (x : type) P(x))** is expanded to **P(O1) and ...and P(On) and CWA-Object(type, O1, ..., On)**; similarly for existential statements. The repair strategy for **CWA-Object** replaces the assumption by inserting an event that creates a new object of **type** that satisfies the constraints of the quantified statement. For example, adding the goal **(Exists (ru : rock-unit) Is-Limestone(ru))** indicates that some limestone formation existed at one time in the region of Figure 1. The debugger can achieve this goal by introducing an event that deposits a limestone formation, followed by an event that erodes the formation away since limestone does not appear in Figure 1a.

Typically, bugs depend on a large number of assumptions, so many repairs are suggested for each bug. Best-first search is used to help control the debugger. The debugger evaluates the global effects of each repair to focus on the most promising hypothesis. The primary component of the evaluation heuristic is the number of remaining bugs, including any unachieved top-level goals. The secondary component, used to differentiate hypotheses with the same number of remaining bugs, is the number of events, the idea being to prefer simpler hypotheses. This is a reasonable metric since our planning/interpretation task involves finding one plausible solution and the number of remaining bugs is often a good measure of the closeness to solving the problem.

The evaluation heuristic uses a technique that finds and incrementally updates the closed-world assumptions that change as a result of a bug repair. The technique is similar to the causal simulation technique used by the GTD tester, but is extended to handle non-linear hypotheses.

3 Debugging an Interpretation

This section describes the complete behavior of our debugging algorithm for the buggy hypothesis of Figure 2. The hypothesis has two bugs — the orientations of **SS1** and **B2** are both 17°, not 12°. Starting first with the bug that the orientation of **SS1** is 17°, the debugger locates the 17 underlying assumptions in the dependency structures of Figure 3 and regresses the desired value of 12° back through the dependencies. Of these assumptions, six are ignored by the debugger because they are considered to be unchangeable — **SS1.orientation@Plan-end=12°** because it is a goal, the values 12°, 5° and 0 because they are constants, and the ordering **Tilt2.end** < **Plan-end** because hypothesized events must occur before **Plan-end**.

For the three **CWA** assumptions, the debugger proposes the same basic repair of adding a new tilt event of -5° between the start and end of the persistence interval. The repairs, however, are rated differently. The evaluation heuristic determines that adding the new tilt between **Tilt2.end** and **Plan-end** repairs both bugs in the initial hypothesis but also introduces two new bugs — the orientations of **SH1** and **B1** are now zero, not 5°. Adding the tilt between **Deposition1.end** and **Tilt1.start** is considered a solution since it repairs both bugs without introducing new ones. Adding the tilt between **Tilt1** and **Tilt2** produces a non-linear hypothesis where the new tilt and **Deposition2** are unordered. This repair is also regarded as a solution since one of the possible linearizations (where the new tilt precedes **Deposition2**) interprets the region correctly.

For the **Occurs(tilt, Tilt2)** assumption, deleting the tilt event fixes the bug, since without **Tilt2** the orientation of **SS1** is 12°. This repair does not solve the whole problem, however, since it introduces the same two new bugs as above. For the other two **Occurs** assumptions, deleting the events does not fix the bug. For all three assumptions, replacing events is not an applicable strategy since our geologic models do not contain events that are similar enough to tilting or deposition.

The two **CWA-Exists** assumptions yield the same basic repair — an erosion event is proposed to destroy the **SS1** formation. The evaluation heuristic rates these repairs poorly, however, since they undo all the goals of the problem by destroying all existing formations and boundaries. The reordering strategy does not succeed for the assumption **Tilt1.end** < **Tilt2.start** because the desired value of **SS1.orientation@Tilt2.start** (12°) is not achieved at the start of **Tilt1**; similarly for the assumption **Deposition1.end** < **Tilt1.start**.

The debugger proposes replacing **Parameter-of(Tilt1, Theta, 12°)** by the assumption that **Theta** is equal to 7°, the difference between the desired value of **SS1.orientation** at the end of **Tilt1** and its predicted value of zero at the start of the event. The evaluation heuristic determines that this repair is a solution. Similarly, for the assumption **Parameter-of(Tilt2, Theta, 12°)**, the debugger considers changing the parameter value to the difference between the desired value of **SS1.orientation@Tilt2.end** (12°) and its predicted orientation at **Tilt2.start** (also 12°). This repair is rejected, however, since the debugger determines that it is inconsistent with a constraint in our domain models that **Theta** must be non-zero.

Thus, the debugger suggests seven potential repairs for this problem, of which three are considered solutions. The evaluation heuristic prefers the repair in which the parameter of **Tilt1** is altered, since this produces an hypothesis with fewer events than the other two solutions, both of which add new tilt events. The preferred solution is the same as in Figure 1b.

4 Completeness, Coverage and Efficiency

By "completeness" we mean can the debugger fix all bugs describable within its representation language? For the assumptions made explicit in our causal models, the dependency tracing and repair strategies are complete, in that they can find all ways to replace assumptions to achieve a desired value.

One caveat is that the strategies cannot in general find repairs that involve replacing multiple assumptions, where changing any one of the assumptions separately has no discernible effect on repairing the bug (*it can* handle situations where more than one assumption is faulty, as long as replacing at least one assumption moves the hypothesis closer to achieving the desired value). For example, the debugger cannot handle situations where a bug depends on two parameters being above a certain threshold, but changing either parameter alone moves the hypothesis further from repairing the bug. One area for future research is to develop general repair strategies that can handle such combinations of assumptions.

Another problem is that the regression technique, while sufficient for the problems we explored, is not theoretically complete due to the difficulty of inverting general functions. If the constraints produced by the regression are not sufficient to determine parameter values precisely, the debugger must choose and test different values until one is found that solves the problem. Incompleteness arises when only a finite number of values out of an infinite set (e.g., the reals) can solve the problem. For example, if a solution depends on a parameter value being exactly π, in general it will take infinite time to test each choice before hitting on the correct solution. This observation also shows that even the simple technique of enumerating and testing all hypotheses is incomplete, since it is not possible to enumerate all hypotheses (in particular, the parameter bindings of events) in finite time.

"Coverage" refers to how well the assumptions handled by our repair strategies cover the range of possible bugs. We note that to provide complete coverage, the debugger must handle all the assumptions needed by the tester to predict effects and detect bugs: the assumptions made in specifying hypotheses, the assumptions about the initial and goal states, closed-world assumptions made by the tester, and assumptions about the correctness of domain models. We argue that our debugger has wide coverage, since it currently handles all but the latter assumption and some types of closed-world assumptions implicit in the tester's algorithms.

For the planning/interpretation task, hypotheses are completely specified by the events that occur, their parameter bindings, and the temporal orderings between events. Thus, pragmatically, these are the only types of assumptions made in constructing hypotheses that need to be handled. Our current debugger has repair strategies to cover each of them. Assumptions about the initial and goal states do not need to be handled — our task specifies that they are unchangeable, since changing the initial and goal states constitutes solving a different problem.

The debugger currently handles three types of closed-world assumptions that are commonly made (and are commonly at fault) in our domains. Practical experience has not shown the need for handling others, although it is a fairly simple matter to extend the debugger to handle other closed-world assumptions by making them explicit in the dependency structures and adding repair strategies for them.

Not so simple to handle are the assumptions that the domain models are correct. Handling them is somewhat tricky because any bug can be fixed by changing the models in an appropriate way. For example, we could debug the example in Section 3 by changing the definition of tilt so that its effect was not uniform for all rock-units. Clearly any reasonable repair strategy that changes domain models must constrain the problem, for instance, by reference to a meta-theory of the domain or by induction using multiple examples, subjects well beyond the current scope of our research.

One downside of our debugging algorithm is its high computational cost. Although each individual repair strategy is fairly efficient, the number of assumptions underlying bugs tends to grow exponentially in domains, such as geology, with many potential interactions among events. In addition, the evaluation heuristic is very expensive since determining the number of remaining bugs is, in general, exponential for the types of non-linear hypotheses produced by our debugger (see [Chapman]). It is these computational reasons that led us to develop the GTD paradigm in which the robust, but slow, debugger is used only to focus on the problems handled incorrectly by the heuristic generator.

5 Relations to Other Debuggers and Planners

The approach of tracing faults to underlying assumptions has roots in work on dependency-directed search (e.g., [Stallman]), model-based diagnosis (e.g., [Hamscher], [deKleer]), and algorithmic debugging [Shapiro]. Our contribution to the dependency tracing approach is in providing principled strategies that determine how to replace the underlying assumptions once they have been located.

Our *assumption-oriented* debugging approach stands in contrast to other approaches in which repair heuristics are associated either with bug manifestations (e.g., [Alterman], [Marcus]) or with certain stereotypical patterns of causal explanations (e.g., [Hammond], [Sussman]). Our approach handles the large number of possible ways bugs can arise by decomposing them into combinations of a small set of underlying assumptions. This approach tends to give greater coverage and also tends to suggest more alternative repairs than other approaches since we do not have to anticipate all possible patterns of assumptions that can lead to bug manifestations.

For example, consider the "Prerequisite Clobbers Brother Goal" bug type in [Sussman] that occurs when an event **X**, in attempting to achieve the preconditions of an event **Y**, undoes a goal that had been achieved by event **Z**. The only repair for this bug type given in [Sussman] is to reorder events **X** and **Z**. [Hammond] presents a similar bug type that has an additional strategy of replacing **Y** with an event that does not have the offending precondition. Our debugger would suggest even more repairs, including

inserting an event to reachieve the goal, replacing event **X**, and changing **X**'s parameters so as to make the goal and precondition true simultaneously.

Our basic debugging strategy — repair one bug at a time by analyzing domain models and then evaluating how the local repair affects the hypothesis as a whole — is similar to the approach used by domain-independent planners (e.g., [Sacerdoti], [Wilkins], [Chapman]). In fact, we can use our debugger as a planner by starting with the null hypothesis and treating all the unachieved, top-level goals as bugs. A major difference, however, is that most domain-independent planners use hypothesis *refinement*, in which the system can only add information to its current hypothesis, making plans increasingly more detailed. Our debugger uses a *transformational* approach, in which information may be deleted as well to change previous decisions made in solving the problem.

The transformational approach is particularly beneficial in complex, relatively underconstrained domains, since the problem solver can make simplifying assumptions and commitments in order to increase problem solving efficiency, with the understanding that erroneous choices can be subsequently debugged. In such domains, refinement and its concomitant strategy of least-commitment are often very inefficient due to the expense of evaluating partially specified hypotheses.

6 Summary

Our theory of debugging involves tracing bug manifestations back to the underlying assumptions, made during hypothesis construction and testing, upon which the bugs depend. The direction in which to change assumptions is indicated by regressing values and constraints back through dependencies. Bugs are repaired by replacing assumptions using a small set of domain-independent repair strategies that reason about the dependency structures, regressed values, and domain models that encode the effects of events. The proposed repairs are then evaluated to determine their overall goodness.

Our theory of debugging provides a very robust framework for repairing bugs in plans and interpretations. The debugging algorithm is nearly complete and the six implemented repair strategies provide good coverage of the common types of faulty assumptions. In addition, the framework is easily extended to handle assumptions currently not made explicitly. A subject for future work is to examine how well the theory extends to debugging in other tasks, such as design or diagnosis, that use different causal models and have different task specifications.

Our approach subsumes earlier work in debugging by using principled assumption-oriented repair strategies to cover more bug manifestations and to suggest more potential repairs for each bug. The transformational approach used by our debugger also extends the refinement approach used by most domain-independent planners. The transformational approach can increase problem solving efficiency by enabling the problem solver to make simplifying assumptions that the debugger can replace if incorrect.

The assumption-oriented debugging approach is still quite computationally expensive, due to the large number of assumptions underlying each bug and the expense of evaluating each proposed repair. We achieve overall efficiency using the Generate, Test and Debug paradigm in which heuristic rules are used to generate an initial hypothesis that is debugged if it turns out to be incorrect.

Acknowledgments

Helpful contributions to this paper were made by Randy Davis, Walter Hamscher, Drew McDermott, Howie Shrobe, and Reid Smith. This work was supported by Schlumberger and the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

References

[Alterman] R. Alterman, An Adaptive Planner, AAAI-86, Philadephia, PA.

[Chapman] D. Chapman, Planning for Conjunctive Goals, *Artificial Intelligence*, vol. 32, pp 333-377, 1987.

[deKleer] J. deKleer, B. Williams, Diagnosing Multiple Faults, *Artificial Intelligence*, vol. 32, pp 97-130, 1987.

[Hammond] K. Hammond, Explaining and Repairing Plans That Fail, IJCAI-87, Milan, Italy.

[Hamscher] W. Hamscher, R. Davis, Issues in Model Based Troubleshooting, AI-Memo 893, MIT, 1987.

[Marcus] S. Marcus, J. Stout, J. McDermott, VT: An Expert Elevator Designer, AI Magazine, vol. 9, no. 1, Spring 1988.

[McAllester] D. McAllester, An Outlook on Truth Maintenance, AI Memo 551, MIT, 1980.

[Sacerdoti] E. Sacerdoti, *A Structure for Plans and Behavior*, American Elsevier, 1977.

[Shapiro] E. Shapiro, *Algorithmic Program Debugging*, MIT Press, 1982.

[Simmons] R. Simmons, Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems, PhD dissertation, AI-TR-1048, MIT, 1988.

[Stallman] R. Stallman, G. Sussman, Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence*, vol. 9, 1977.

[Sussman] G. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier, 1977.

[Wilkins] D. Wilkins, Domain-Independent Planning: Representation and Plan Generation, *Artificial Intelligence*, vol. 22(3), pp 269-301, 1984.