Technical Report 1048

Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems

Reid Gordon Simmons

MIT Artificial Intelligence Laboratory

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI TR 1048	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Combining Associational and Causal Reasoning to Solve Interpretation and Planning Problems		5. TYPE OF REPORT & PERIOD COVERED technical report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Reid Simmons		8. CONTRACT OR GRANT NUMBER(s) N00014-85-K-0124
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE September 1988
		13. NUMBER OF PAGES 214
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

associational reasoning	geologic interpretation
causal reasoning	debugging
planning	multiple representations and reasoning techniques

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Efficiency and robustness are two desirable, but often conflicting, characteristics of problem solvers. This report presents an approach, called Generate, Test and Debug (GTD), that integrates associational and causal reasoning techniques to efficiently solve a wide class of interpretation and planning problems.

COMBINING ASSOCIATIONAL AND CAUSAL REASONING TO SOLVE INTERPRETATION AND PLANNING PROBLEMS

by
REID GORDON SIMMONS

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
SEPTEMBER 1988

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on May 10, 1988 in partial fulfillment of the requirement for the degree of Doctor of Philosophy.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the author was provided in part by grants from Schlumberger-Doll Research Labs and the National Science Foundation, and by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-85-K-0124.

Abstract

Efficiency and robustness are two desirable, but often conflicting, characteristics of problem solvers. This report presents an approach, called Generate, Test and Debug (GTD), that integrates associational and causal reasoning techniques to efficiently solve a wide class of interpretation and planning problems.

The GTD paradigm generates an initial hypothesis using rules that associate features of the problem with events that can cause them. If the tester detects bugs in the hypothesis, it is debugged until a correct solution is produced. The debugger employs three domain-independent causal reasoning techniques: 1) it analyzes causal explanations produced by the tester to locate the assumptions underlying bugs in the hypothesis, 2) it regresses values back through the explanations to indicate the direction in which to change the assumptions, and 3) it replaces faulty assumptions based on a model of causality that explicitly represents time, persistence, and the effects of events.

Our analysis of the GTD paradigm indicates that the generator's efficiency stems from its use of nearly independent associational rules. This enables it to construct hypotheses that are correct, or nearly so, without having to check for potential interactions between events. In contrast, the debugger achieves robustness by using causal models of how the world works to determine how events interact in the achievement of goal. We characterize domains for which GTD may be useful based on this analysis of the strengths and weaknesses of the two reasoning techniques.

An implementation of GTD has been tested in several domains, including our primary domain of geologic interpretation, blocks-world planning, and the Tower of Hanoi problem. In addition, parts of the system have been used to help diagnose manufacturing faults in semiconductor fabrication.

Acknowledgements

During my brief tenure at MIT (brief, at least, by geologic standards) many people have contributed to making it an intellectually and personally rewarding experience. Foremost has been my advisor Randy Davis who, by word and deed, taught me how to do research critically and express the results intelligibly. He was always available to comment on my ideas and writings, invariably giving sound advice (even if it was not always heeded).

Members of the hardware-troubleshooting research group — Walter Hamscher, Brian Williams, Mark Shirley, and Jeff Van Baalen, in particular — have provided much intellectual stimulation and friendship. It is a privilege to have been associated with them over the years. My committee members, Chuck Rich and Peter Szolovits, supplied crucial insights into what was important and interesting about the research. I have also benefited from discussions with Phil Agre and David Chapman.

I am grateful to Schlumberger for their support, both financially and intellectually, during the course of my research. In particular, I have benefited much from interactions with Reid Smith, Marty Tenenbaum, John Mohammed, and Roy Nurmi, who taught me to do geologic interpretation. Many others outside the MIT community have had an impact on my work, including Bruce Buchanan, Drew McDermott, Tom Dean, Yoav Shoham, Kris Hammond and Ken Forbus. To these and many others, my heartfelt thanks.

Most importantly, I thank my dear wife Pearl, who is a continual source of love, friendship and amusement. She has been a source of strength and comfort when I needed it the most. I eagerly anticipate embarking on this new phase of my life with her.

Contents

1. Introduction

Efficiency and robustness (the ability to solve a wide range of problems) are two desirable, but often conflicting, characteristics of problem solvers. This report describes an approach that combines associational and causal reasoning techniques to achieve both efficiency and robustness. The approach uses the Generate, Test and Debug (GTD) paradigm: the generator uses associational reasoning to construct an initial hypothesis that is typically correct or nearly so; the tester simulates the hypothesis to determine whether it is in fact correct; the debugger uses causal reasoning to repair any faulty hypotheses found.

We have explored the use of GTD primarily for tasks of the form "given an initial state and a final (goal) state, both of which are partial descriptions of states of the world, find a set of events that could achieve the final state from the initial state." Both interpretation and planning problems fit this task specification. If the final state is in the future, we regard it as a planning problem; if the initial state is in the past, we regard it as interpretation.

GORDIUS,[1] 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 addition, some experiments have been performed using GORDIUS to diagnose manufacturing faults in semiconductor fabrication.

Associational reasoning techniques solve problems by associating observable features in the input states with events that could have caused those features. For example, if we observe one sedimentary formation on top of another, we can associate this with the likely explanation that the lower formation was deposited before the upper one. The GTD generator uses associational rules that are nearly independent, which enables it to create hypotheses that are usually correct without having to reason about potential interactions between events.

Causal reasoning techniques solve problems by analyzing the cause and effect relationships of events. For example, given the same two

[1] GORDIUS was the king who tied the Gordian Knot. Undoing it was an intricate problem that could be solved only by cutting through the knot.

sedimentary formations as above, we can causally analyze our model of deposition to determine that deposition always happens from above, adding material along the surface of the Earth. Thus, to be below the upper formation, the lower one must have already existed on the surface at the time the upper formation was deposited. Hence, the lower formation must have been created earlier. The debugger determines how to repair bugs by reasoning about the effects of events, interactions between events, and the changes that happen to objects over time.

In short, associational reasoning is efficient because it uses features of the problem to focus in on possible solutions and it composes partial solutions without reasoning about interactions. Causal reasoning is robust because it analyzes causal models of how the world works to focus in on the events needed to change the world to achieve the desired results. The integration of these two techniques gives GORDIUS a high degree of performance and competence. In addition, our understanding of their strengths and weaknesses enables us to characterize domains for which the GTD paradigm may be useful.

Much of our research has focused on developing the representations and causal reasoning techniques used by the tester and debugger. A wide class of events can be represented, including those that create and destroy objects and those that have conditional, quantified, and relative effects. To predict the effects of events accurately and efficiently, GORDIUS employs several specialized representations for reasoning about quantities, sets, diagrams, time, and the persistence of attributes and objects over time. These representations are integrated into a common framework, enabling inferences made in one representation to constrain others.

We have developed a theory of debugging in which bugs are repaired by tracking down and replacing faulty assumptions made during the generation and testing of hypotheses. Faulty assumptions are replaced using domain-independent repair strategies that reason about the type of assumption, causal explanations for why the bugs arose, and causal domain models. For example, if a bug arises because the preconditions of an event are not met, the debugger tries to replace the event with one that has the same desirable effects but avoids the offending precondition. The debugger then reasons about interactions between events to determine the effect of each repair on the hypothesis as a whole.

1.1 The GTD Paradigm

The GTD paradigm was developed to take advantage of the different strengths of associational and causal reasoning. The generator efficiently produces solutions to most problems. For those problems that it handles incorrectly, the more robust, but computationally expensive, debugger is used to transform the generator's initial hypothesis into a solution.

Input to the GTD paradigm are descriptions of the initial and final states of the problem. The generator matches a library of associational rules against these state descriptions and combines matching rules to construct an initial hypothesis that purports to explain how the final state could be achieved from the initial state. This hypothesis is then tested using simulation techniques. If the result of the simulation matches the final state, the hypothesis is accepted as a solution; otherwise, the tester passes to the debugger causal explanations for the bugs detected.

The debugger suggests modifications to the hypothesis to repair each bug. It then estimates the global effects of the modifications — whether they introduce new bugs or achieve any of the remaining ones. When satisfied that all bugs have been repaired, the debugger submits the modified hypothesis 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.

Each stage of the paradigm has a different task and requirements. The generator must efficiently produce hypotheses that are usually correct or nearly so. The tester must be accurate enough to determine whether an hypothesis actually solves the problem. The debugger must be robust enough to repair any bugs detected by the tester. To achieve these desired characteristics, each stage uses reasoning techniques and representations appropriate to its particular task.

The representations used by the generator are domain-specific rules, which we call *scenarios*, that associate patterns in the initial and final states with events that plausibly explain how the patterns could be achieved. In the geologic domain, for instance, the pattern of an igneous rock surrounded by two rocks of the same composition is likely to have arisen from the igneous rock intruding (pushing) through an existing rock, splitting it into two pieces.

The generator works by matching scenario patterns against the input state descriptions and composing the events from different matching scenarios to generate a complete hypothesis. In essence, the matching patterns indicate sub-problems and the associated events are sub-solutions. Events are composed by trying to unify them using simple, but general, principles — two events unify if they are of the same type and have consistent parameter bindings. Unifying events together indicates that they are really the same event. For example, two events that purport to create pieces of the same rock must be the same event, since objects can be created by only one event.

The generator achieves efficiency by, for the most part, independently composing the events of different matching scenarios. It presumes, in other words, that the events associated with one scenario pattern do not interfere with the achievement of other patterns. Under this presumption of composability, hypotheses can be constructed in time proportional to the number of goals.

Unfortunately, the generator may produce incorrect hypotheses since the presumption of composability does not necessarily hold for all combinations of scenarios. Thus, GTD tests the generated hypotheses, simulating them to verify whether they are correct. The GORDIUS tester relies on a domain-independent causal simulator to predict the changes that the hypothesized events have on objects in the domain. The simulator reflects these changes by updating the various specialized representations of the system, using event models that declaratively represent their preconditions and effects.

For geologic interpretation problems, an additional diagrammatic simulation is used to predict the spatial effects of events more accurately. The diagrammatic simulator constructs a sequence of diagrams to reflect geometrical and topological changes to the geologic objects. A specialized diagram matcher, which uses a Waltz-type filtering algorithm, is used to compare the topological structure of diagrams to determine whether the simulation diagram matches the goal state of a geologic interpretation problem.

The main task of the tester is to detect bugs in the hypothesis, where a bug is an inconsistency between the desired value of a statement and its value as predicted by the tester. For example, a bug arises if the desired orientation of a formation in the final state is 10°, while the orientation predicted by simulating the hypothesis is 5°. The tester detects two types of bug manifestations: 1) the preconditions of an event do not hold, and 2) the set of events does not achieve the goal state. For each bug detected, the tester constructs a *causal dependency structure* to explain why the bug arises. The dependency structures, which are networks that represent causal, functional and logical dependencies between nodes, are produced as a by-product of our causal simulation algorithm.

The debugger repairs bugs using three causal reasoning techniques: 1) it analyzes the causal dependency structures produced by the tester to find the assumptions that underlie the explanations for why the bugs arise, 2) it regresses values back through the dependency structures to determine the desired values that need to be achieved, and 3) it uses repair strategies that know how to replace faulty assumptions with ones that can achieve the desired values.

For example, if a bug depends on the assumption that a parameter has a particular value, the general repair strategy is to change the parameter to a value that fixes the bug. This desired parameter value is determined by regressing the value that fixes the bug back through the causal dependency structure. As a simple example, suppose the desired orientation of some formation **S1** is 14°, while the tester predicts that the hypothesized sequence of events "first deposit **S1** horizontally, tilt all rock-units by 7°, then tilt again by 5°" will produce a value of 12°. By regressing the value of 14° back through the causal explanation for the bug, the repair strategy determines that changing the parameter value of the first tilt event from 7° to (14-5)° repairs the bug.

For each proposed bug repair, the debugger determines how the repair interacts with the hypothesis as a whole by using an evaluation heuristic to estimate the number of bugs remaining in the hypothesis. The evaluation heuristic uses a technique that is similar to the causal simulator of the tester, but is somewhat less computationally expensive and less complete.

In GORDIUS, best-first search is used to control the overall search for a solution. When a partial hypothesis is proposed, either by composing events from a matching scenario or by repairing a single bug, the hypothesis is placed on a priority queue sorted by an estimate of the closeness of the hypothesis to a solution. GORDIUS uses a fairly simple distance metric whose primary component is the number of bugs in the hypothesis and whose secondary component is the cost of the hypothesis, estimated as the number of hypothesized events. This distance metric captures our desire to find a solution that is both complete (i.e., it achieves all the goals) and compact, in the sense of Occam's Razor. In practice, we have found the metric to be quite adequate: for most problems examined, this distance metric results in a very focused search, with few false paths pursued.

Two aspects of our search strategy should be noted. First, using one queue to keep track of hypotheses proposed by both the generator and debugger provides a simple way to decide dynamically when to continue debugging and when to return to the generator for a new hypothesis. Under best-first search, control returns to the generator whenever the next best generated hypothesis has fewer unachieved goals than any of the hypotheses being debugged. Second, the search heuristic does not attempt to optimize over any aspect of the hypothesis since our task is to find a plausible, although not necessarily the best, solution. While the tendency is to produce solutions with fewer events, no guarantee is made that the solution found will be the shortest.

1.2 Domains Explored

The GTD paradigm was originally developed for solving geologic interpretation problems. The task is to infer a set of events that can plausibly explain how a geologic region was formed, given a vertical cross section of the region.

A cross section (the problem's goal state) describes topological, geometrical, and compositional aspects of the region. The goal state is shown schematically as a two-dimensional diagram and a legend identifying the rock types (Figure 1). The initial state, which is the same for all geologic interpretation problems we considered, consists simply of bedrock under sea-level.

A solution to a geologic interpretation problem consists of the types of events that occurred and constraints on the temporal orderings and parameter bindings of the events (e.g., "there is enough erosion to partially erode the shale"). Figure 2 presents a plausible solution to the problem in Figure 1. Unless otherwise noted, GORDIUS has solved all examples presented in this report, and all the geologic diagrams, such as Figure 1, are produced by the system.

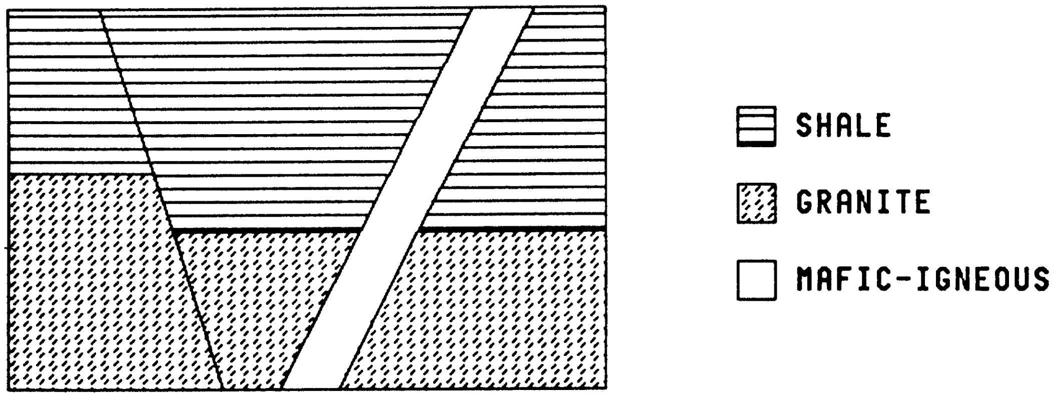


Figure 1. Geologic Interpretation Goal State

1. Deposit Shale
2. Intrude Granite into Shale
3. Uplift
4. Intrude Mafic Igneous through Granite and Shale
5. Fault across Shale and Granite
6. Erode Shale and Mafic Igneous

Figure 2. Plausible Solution Sequence to Figure 1

Geologic interpretation is a basic skill needed by geologists. It is often the first step in analyzing geologic data, since the types and ordering of events that occurred are crucial in answering such questions as whether oil or minerals can be found in a region. It is also one of the first skills taught to geologists. In fact, several of our interpretation problems, including Figure 1, are adapted from midterm examinations given to first-year geology undergraduates.

Geologic interpretation is a good domain for studying problem solving. Knowledge acquisition is not a bottleneck since the problem requires relatively little geologic knowledge, as evidenced by the fact that first-year students are readily taught to perform the task.

Qualitative, commonsense models of geology and simple quantitative models suffice for doing many interpretation problems.

On the other hand, modeling geologic events involves representing and reasoning about a wide range of temporal, spatial, conditional and quantified effects, including the creation and destruction of objects. The large number of potential interactions among events means that problem solvers employing purely associational reasoning will tend to be brittle, since associational reasoning alone cannot handle unforeseen interactions. Similarly, purely causal reasoning is computationally infeasible in this domain due to the cost of determining all potential interactions. These domain characteristics indicate the need for a problem-solving paradigm that combines the best of both types of reasoning.

We have also carried out less extensive exploration of several other domains: blocks-world planning, the Tower of Hanoi problem, and semiconductor fabrication diagnosis. Blocks-world planning was chosen because it is a well-studied domain in AI and serves as a testbed to compare our methods with other planning techniques. We have found that GORDIUS compares very favorably with traditional domain-independent planners (e.g. [Chapman], [Sacerdoti], [Sussman], [Wilkins]).

The Tower of Hanoi problem was chosen because it is characterized by a high degree of interaction — moving a ring from one post to another is likely to interfere with the goals of moving other rings. As we will see, the degree of interaction has a profound effect on the problem-solving efficiency of the GTD paradigm. Finally, we briefly explored a rather different domain, that of diagnosing manufacturing faults in semiconductor fabrication [Mohammed & Simmons]. This domain was chosen primarily to exercise the Test and Debug algorithms in another complex, real-world domain.

1.3 A Geologic Interpretation Problem

This section briefly describes how GORDIUS solves the geologic interpretation problem of Figure 3 in order to introduce the reasoning techniques and knowledge used in GTD. The details of these techniques are discussed in Chapters 2, 3, and 4.

The three basic types of geologic objects in our model are rock-units, boundaries, and points. A rock-unit is simply a mass of rock. A formation is a rock-unit of homogeneous composition that was formed by a single event. A boundary is the interface between two rock-units or between a rock-unit and the **Environment** (the air or sea). A fault, for example, is represented as the boundary between the rock-units on either side of the fault (called the up-thrown and down-thrown blocks). The surface of the Earth is the boundary between the **Environment** and the top-most rock-units of a region. A point, the third type of geologic object, represents a location on a rock-unit or along a boundary. For example, "the top of the shale formation" and "the bottom of the Earth's surface" are both points. In addition, rock-units and boundaries may be composed of "pieces" of other objects of the same type. For example, the shale formation in Figure 3 is composed of the shale pieces **SH1**, **SH2** and **SH3**, and the surface of the Earth is composed of pieces **B1-B4**.[2]

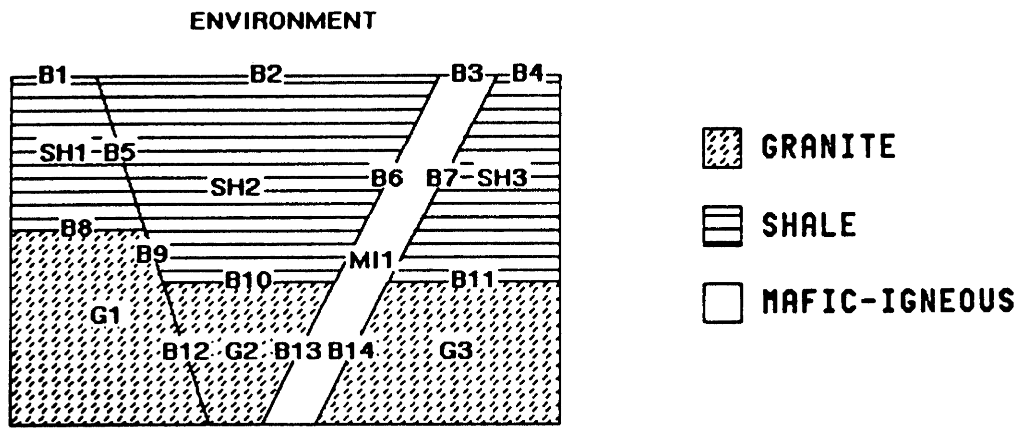


Figure 3. Geologic Interpretation Problem

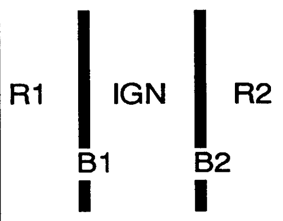
We have employed a simple model of geology known as "layer cake" geology to model seven of the most common types of geologic events — deposition, erosion, intrusion, faulting, uplift, subsidence, and tilt. In our model, deposition, which occurs when silt in water deposits on the sea bed, creates horizontal sedimentary formations (e.g., shale and sandstone) that stack up like the layers of a cake. Erosion, which occurs when wind abrades exposed rock formations,

[2] The edges along the sides and bottom of the diagram form its *window*. Although the system has no information about the appearance of the region outside the window, it is presumed to continue linearly.

is modeled as occurring horizontally, slicing through the Earth like a knife. Intrusion creates igneous formations (e.g., mafic-igneous and granite) when molten rock from below intrudes (pushes) into or through upper rock layers. We distinguish two types of intrusion — dike-intrusion and batholithic-intrusion. Dike-intrusion occurs when a thin band of igneous rock intrudes all the way through existing rock-units. **MI1** in Figure 3 is a dike-intrusion. Batholithic-intrusion occurs when a much larger mass of igneous rock intrudes into, but not through, rock-units. **G2** in Figure 3 is a piece of a batholithic-intrusion. Faulting splits the Earth and, in our model, moves one side of the fault (the down-thrown block) downward relative to the other side (the up-thrown block). Uplift and subsidence move the Earth uniformly up or down, respectively; tilt rotates the Earth around some origin.

GORDIUS begins solving the problem of Figure 3 by matching *scenarios* against the goal diagram and combining matching scenarios to generate an initial hypothesis. Scenarios are heuristic rules that associate observable patterns in the goal and initial states with sets of events, called *local interpretations*, that could have produced the patterns. For example, the "intrudes-through" scenario, illustrated schematically in Figure 4, indicates that an igneous formation intruded into an existing rock formation. The scenario pattern (Figure 4a) matches those parts of a goal diagram where an igneous rock is between two rocks of the same composition and the boundaries of the rocks are parallel. One match in Figure 3 occurs where **MI1** is between **SH2** and **SH3**.

A. Schematic Representation of Scenario Pattern



Pattern Constraints:
Igneous(IGN)
Same-type(R1, R2)
Parallel(B1, B2)

B. Local Interpretation

Events:
1. Create Rock1
2. Intrude IGN through Rock1

Interpretation Constraints:
Piece-of(Rock1, R1)
Piece-of(Rock1, R2)
Intersects(Rock1, IGN)

Figure 4. Schematic of the "Intrudes-Through" Scenario

The local interpretation describes the events that occurred to form the pattern, their parameter bindings, temporal orderings between events, and the objects that are related via a piece-whole hierarchy. The local interpretation of the "intrudes-through" scenario (Figure 4b) indicates that one event creates formation **Rock1** (where **R1** and **R2** are pieces of **Rock1**), which is followed by a dike-intrusion event that creates the igneous formation **IGN**. In addition, the local interpretation constrains the location of the igneous formation to intersect with the existing rock formation.

Note that a scenario is just a heuristic since different events may give rise to the same pattern. For example, there are at least two interpretations for the pattern of a sedimentary rock on top of an igneous rock: 1) the igneous rock intruded into the sedimentary rock, or 2) the igneous rock intruded into some pre-existing rock, everything was uplifted, the upper layers eroded exposing the igneous rock, after which the region subsided and sedimentary rock was deposited on top of the igneous rock. Although the generator prefers to use the first interpretation since it is shorter, the second may be used if the first leads to an inconsistency.

The local interpretations derived from matching scenarios are combined to yield an initial hypothesis. In generating the initial hypothesis shown in Figure 5, thirteen matches are employed. The generator uses six of the fifteen currently defined scenarios (gi-scenarios-fig), with some of the scenarios, such as "intrudes-through", matching in more than one place.

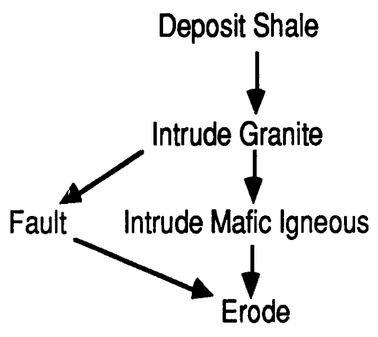


Figure 5. Initial Hypothesis Sequence Generated

The hypothesis of Figure 5 differs from the solution in Figure 2 in two important respects. First, the generator leaves the faulting and intrusion events unordered since the goal diagram does not contain enough information to tell which occurred first. Second, the hypothesis of Figure 5 does not contain an uplift event because the "erosion-ends-boundary" scenario was altered for this example in order to provide a simple illustration of the test and debug capabilities of GTD.

This initial hypothesis is then tested using a combination of causal and diagrammatic simulation techniques [Simmons, 83]. To make the testing more efficient, GORDIUS first simplifies the initial hypothesis by choosing one of the totally ordered sequences consistent with it. This is a reasonable simplification because our task is to produce one plausible interpretation; if the wrong linearization is chosen, GORDIUS can use the debugger to repair the choice.

The causal simulator takes each event in the sequence in turn and checks whether the preconditions of the event hold. If so, the simulated state of the world is updated to reflect the effects of the event. In this example, the tester proceeds until the erosion step is reached, at which point a bug is detected. The bug is that erosion cannot occur because , being a wind-driven event, it must occur above sea-level, but the current state of the simulation indicates that the surface of the Earth remains below sea-level, unchanged since the deposition of shale.

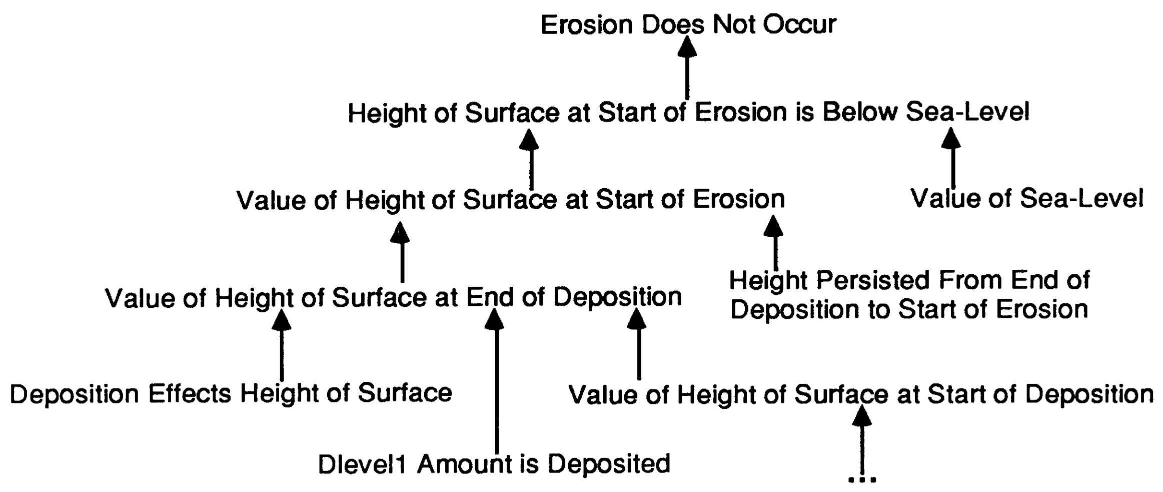

Erosion Does Not Occur
Height of Surface at Start of Erosion is Below Sea-Level
Value of Height of Surface at Start of Erosion
Value of Sea-Level
Value of Height of Surface at End of Deposition
Height Persisted From End of Deposition to Start of Erosion
Deposition Effects Height of Surface
Value of Height of Surface at Start of Deposition
Dlevel1 Amount is Deposited
...

Figure 6. Causal Dependency Structure for the Bug that Erosion Does Not Occur.

The tester passes to the debugger a dependency structure, outlined in Figure 6, that explains how the events caused the bug (an arc means "depends on"). The debugger traces back through the dependency structure to locate the assumptions underlying the bug, two of which are: 1) the assumption that some fixed amount of material, **Dlevel1**, is deposited, and 2) the closed-world assumption that the height of the surface persisted from the end of deposition to (at least) the start of erosion.

The debugger's repair strategies attempt to replace the assumptions with ones that will fix the bug. For example, taking assumption #1 above, the debugger considers increasing the parameter **Dlevel1** enough to raise the surface of the Earth above sea-level. The debugger infers that this cannot help, however, since our model of deposition indicates that it can only occur under water, hence no amount of deposition can raise the surface above sea-level.

Analyzing assumption #2, the debugger considers replacing the persistence assumption by adding an event that can increase the height of the surface. Two possibilities are uplift and tilt. The debugger evaluates both repairs and estimates that adding an uplift event is better since tilt would introduce new bugs, namely that the orientations of boundaries would no longer match those of the goal diagram. The debugger thus inserts an uplift event between the deposition and erosion events (Figure 7), adding the parameter constraint that the amount of uplift must be enough to raise the surface of the Earth above sea-level.

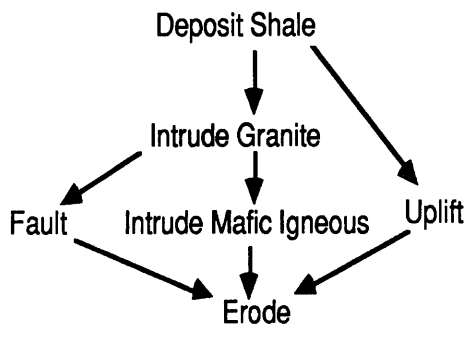


Figure 7. Debugged Hypothesis to Solve Figure 3.

This modified hypothesis is then submitted to the tester for verification. As mentioned above, the tester chooses one total ordering consistent with the partial ordering in Figure 7 (e.g., Figure 2). This time the causal simulation completes successfully. It is followed by a diagrammatic simulation that tests the hypothesis in more detail by constructing a series of diagrams that represent the spatial effects of the geologic events. Since the final simulation diagram (Figure 8) matches the goal diagram (Figure 3), GORDIUS concludes that the sequence of Figure 2 is one plausible interpretation for this problem.

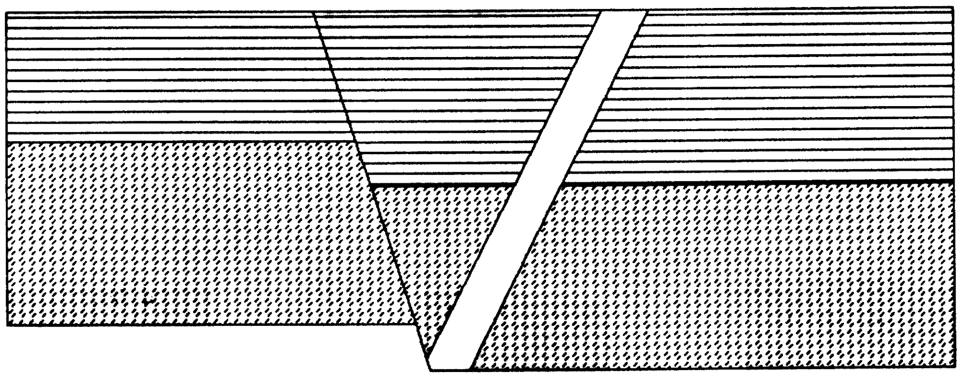


Figure 8. Successful Simulation of Interpretation Example

1.4 History and Contributions

The problem-solving paradigm described in this report has a rich history in AI. The "Problem Solving by Debugging Almost Right Plans" paradigm of [Sussman], the "Abstraction, Inspection and Debugging" paradigm of [Rich & Waters], and much of the work in case-based reasoning (e.g., [Alterman], [Hammond], [Kolodner], [Schank]) all use associational reasoning to generate a correct or nearly correct hypothesis and use causal reasoning, when necessary, to do debugging.

Our research extends this previous work in two major ways. First, we have developed a domain-independent theory of debugging for plans and interpretations, and can characterize its completeness with respect to a set of models underlying the problem solver. Second, we have analyzed why associational reasoning tends to be an efficient problem-solving technique and why causal reasoning tends to be robust, yielding insight into the types of problems amenable to GTD-type problem solvers.

1.4.1 A Theory of Debugging

In describing HACKER, one of the first attempts at problem solving by debugging, Sussman acknowledges that the "bug classifier in HACKER is an *ad hoc* program ... An important area for development ... would be the systematization of the knowledge in Types of Bugs in a more modular form" [Sussman, p. 66]. Our theory of debugging takes a large step in that direction. While an *ad hoc* approach might be sufficient for simple blocks-world problems, the complexity of the geologic domain calls for a more systematic approach. In particular, the many combinations of different types of events, effects, and possible interactions among events make it infeasible to produce a robust debugger by introspectively enumerating possible bug types and repairs.

Our theory of debugging treats bug manifestations, such as precondition violations and failure to achieve goals, as surface indicators of deeper failures. The basic idea is that all bugs ultimately derive from faulty assumptions made during the construction or testing of hypotheses. The underlying assumptions are located by tracing back through causal dependency structures that explains why bugs appear. The direction in which to change an assumption is indicated by regressing the desired value of the bug back through the dependencies. Bugs are repaired by replacing faulty assumptions using domain-independent algorithms that analyze the domain models, dependency structures, and regressed values.

Our *assumption-oriented* theory of debugging is very general since a large number of potential bug types can arise from combinations of a small set of different assumptions. Unlike other debuggers, which use a predefined library of bug types (e.g., [Hammond], [Sussman], [Marcus]), our debugger can handle novel bug types by using general methods to analyze the explanations for why bugs arise. Our approach of tracing faults to underlying assumptions has roots in work on model-based diagnosis (e.g., [Davis], [deKleer & Williams]) and algorithmic debugging [Shapiro]. Our contribution is in providing principled strategies for repairing bugs by replacing the underlying assumptions once they have been located.

The coverage provided by the debugger is characterized by comparing the types of assumptions currently handled with the range of different types of assumptions that could possibly arise. The range of assumptions is determined by analyzing three different models upon which the problem solver is built — a model of causality, a model of the problem-solving task, and a model of hypothesis construction. Our model of causality, which describes how change occurs, indicates that the state of the world can be predicted given assumptions about the initial state, the hypothesis, and certain closed-world assumptions, such as that attributes persist unless some known event changes them. Since a problem is solved when the predicted state of the world matches the final state, the coverage of the debugger is given by how well it can handle the above assumptions.

The problem-solving task model states that the assumptions about the initial and final states, which define a particular problem, are fixed and cannot be changed by the debugger. The model of hypothesis construction indicates that an hypothesis is completely specified by assumptions about 1) the events that occur, 2) parameter bindings of events, and 3) temporal orderings between events; the debugger has repair strategies for each of those.

The debugger also handles the types of closed-world assumptions most commonly at fault in our examples: assumptions that attributes persist, that objects exist until destroyed, and that all objects are known to the system. Other closed-world assumptions, such as those regarding the correctness of the domain models, while beyond the scope of this report should be amenable to debugging strategies similar to those described here.

While the debugger is quite robust, it is not theoretically complete in that it cannot solve all problems describable in its representation language. Although the dependency tracing technique and repair strategies *are* complete, the regression technique is not. Pragmatically, however, this has not affected the debugger's ability to solve problems.

1.4.2 Combining Associational and Causal Reasoning

Our analysis of the problem-solving characteristics of GTD centers on how the different techniques represent and reason about interactions between events. Associational reasoning tends to be an efficient means of problem solving because it presumes that its scenarios encapsulate interactions and can therefore be combined independently. A more robust reasoning technique is needed, however, in cases where this presumption is incorrect. In such cases, causal reasoning is used since it explicitly represents and reasons about interactions between events.

A scenario is said to encapsulate interactions if the events within the scenario are sufficient to achieve its pattern and events outside the scenario do not interfere with the achievement of the pattern. In the blocks-world domain, for instance, the scenario "to achieve **On(x, y)** and **On(y, z)**, put **y** on **z** then put **x** on **y**" encapsulates the interaction that the alternative ordering of first putting **x** on **y** would interfere with subsequently putting **y** on **z**.

The advantage of using scenarios that encapsulate interactions is that solutions can be generated by independently composing solutions to subproblems. This has direct bearing on efficiency, since solving each subproblem independently reduces the need for computationally expensive search [Simon]. In the best case, where the scenarios are totally independent, problems can be solved in time proportional to the number of goals in the problem. For example, we can construct a plan for achieving **On(B, C)**, **On(C, D)** and **On(A, B)** by twice using the blocks-world rule above. One application gives us the fragment "put **C** on **D** then put **B** on **C**" and the second application yields "put **B** on **C** then put **A** on **B**." By unifying the two occurrences of "put **B** on **C**," that is, by assuming that they represent the same event, we come up with the totally ordered solution "put **C** on **D**; put **B** on **C**; put **A** on **B**."

Unfortunately, in many domains it is often impossible to produce rules that are completely independent of one another. For example, our "sedimentary-tilt" scenarios states that a sedimentary rock whose orientation is a non-zero angle θ was formed by deposition (which deposits rocks horizontally) followed by a tilt of θ degrees. While this scenario yields a correct interpretation in many cases, it is not independent for problems in which there are two sedimentary rocks oriented at different angles, for instance, a sandstone oriented at 10° beneath a shale oriented at 5°. Using the above scenario twice results in the interpretation "deposit sandstone; tilt 10°; deposit shale; tilt 5°," which is incorrect because the sandstone ends up with an orientation of 15°.

One way of alleviating the problem of non-independent rules is to create more scenarios, ordered by increasing specificity of the patterns. We might, for instance, encapsulate the above interactions by creating a scenario whose pattern matches two sedimentary rocks that have different orientations and whose local interpretation states that the angle of the first tilt event is the difference between the orientations. This is futile in general, however, since in most complex domains unanticipated interactions will always remain. Our strategy, therefore, is to use scenarios that encapsulate the common patterns of interaction and let the causal reasoning handle residual interactions. We are thus prepared to efficiently handle domains in which most, but not all, interactions are encapsulated.

Causal reasoning achieves robust problem-solving behavior by explicitly representing and reasoning about interactions between events. Events interact when one event affects how another event changes the state of the world. The causal reasoning techniques deal with interactions between events by representing and reasoning about time, persistence, and domain models that encode the effects that events can have.

In this sense, our debugger is like traditional domain-independent planners (e.g., [Chapman], [Sacerdoti], [Vere], [Wilkins]), which are concerned with achieving goals and using critics to find interactions between goals. The causal reasoning techniques used by our debugger extend this work in two important ways. First, the complexity of the geologic domain demands more sophisticated ways to represent and reason about the effects of events. In particular, we need to represent the creation and destruction of objects and conditional, quantified and relative effects.

Second, while traditional domain-independent planners use the refinement approach, in which constraints are added only when absolutely necessary, our debugger adopts a *transformational* approach, in which constraints may be removed from an hypothesis as well. The transformational approach is particularly beneficial in complex, relatively under-constrained domains, such as geology, since the problem solver can increase efficiency by making simplifying assumptions and commitments, with the understanding that erroneous choices can be subsequently debugged.

One downside of using causal reasoning is its high computational cost. In solving a problem, the debugger tends to examine more hypotheses than does the generator and the cost of evaluating each hypothesis is higher — exponential for the debugger vs. polynomial for the generator. It is for precisely this reason that GTD uses the debugger sparingly to focus on the problems that the generator cannot handle.

We have developed guidelines for other domains in which GTD may be useful based on this understanding of the different characteristics of associational and causal reasoning. The most important guidelines are that the goals of the problem should be neither totally independent nor totally interdependent. If the goals are totally independent (or can be factored into totally independent encapsulations) the debugger is not needed since the generated hypothesis will always be correct. On the other hand, if the goals are totally interdependent, the generator will frequently produce incorrect hypotheses, and the system's behavior will be dominated by the computationally expensive causal reasoning. The best domains for GTD fall between the two extremes, with the overall problem-solving efficiency increasing with the degree of independence of the goals. To give a search space analogy, the GTD paradigm works well where the generator can quickly find the right "hill" in a bumpy search space and the debugger can incrementally move up the hill to a solution.

1.5 Terminology

This section describes the terminology and syntax used in this report for describing models of the world, problems, and solutions.

Time is represented both as *points*, designated **t1...tn**, and *intervals*, designated **I1...In** (descriptions of our world models are always designated using **boldface** text). Time intervals are defined by their start and end points, **I.start** and **I.end**, respectively. Relationships between time points are indicated using standard inequality relationships ($<$, $\leq$, $>$, $\geq$, $=$), for instance, **t1 < t2**. Time intervals are related through their start and end points, for instance, **I1** preceding **I2** is represented as **I1.end < I2.start**.

Our world model consists of *objects*, such as rock-units, blocks, and quantities. Objects are typed, with the types forming a strict hierarchy. *Temporal objects* correspond to real-world entities. A temporal object, such as **Rock1**, has a temporal extent that is designated by the object's creation time (**Rock1.start**) and destruction time (**Rock1.end**), which may be unspecified if the object still exists.

A temporal object has a set of attributes determined by its type. For example, the attributes of a rock-unit include its thickness, composition, top-most and bottom-most points. The attributes of temporal objects encode their entire *history* over time; a *temporal reference*, written **Object.attribute@time**, is used to refer to the value of an attribute at a particular point in time. For example, **R1.thickness@t1** represents the thickness of rock-unit **R1** at time **t1**. Temporal references may be nested, for instance, **(USA.president@1988).hair-color@1955** designates the hair color, in 1955, of the person who is president of the United States in 1988. If the same time point appears in a nested temporal reference, the expression can be abbreviated by removing the redundant time point. For example, **(Surface.top@t1).height@t1** and **Surface.top.height@t1** both represent the height of the top of the Earth's surface at time **t1**.

Unlike temporal objects, *abstract objects*, such as quantities, sets, and symbolic constants, do not correspond to real-world entities. Abstract objects have no temporal extent and their values do not change over time. The value of an abstract object is represented without using the "@" notation, for instance, **Q1 > 5** represents that the value of quantity **Q1** is greater than five. The elements of sets are represented either as **A** $\in$ **S1** or by using the brace notation, such as **{A B C}** and **{}** (null set).

A *statement* about the world is constructed by combining more primitive statements (e.g., time points, objects, and temporal references) using functions (e.g., **Q1+Q2**), predicates (e.g., **Clear(A, t1)**), arithmetic relations (e.g., **Q1 < Q2**), and logical connectives (e.g., **P** $\Rightarrow$ **Q** — material implication). For example, the statement **(A.top@t1 = {})** $\Rightarrow$ **Clear(A, t1)** defines that block **A** is clear at time **t1** if its top attribute contains no objects.

The value of a statement is determined by *evaluating* the statement. For relations, predicates, and logical connectives the possible values are **true** and **false**. Functions evaluate to objects or constant symbols. In addition, statements can evaluate to **unknown**. When GORDIUS evaluates a statement, it records dependencies that justify the value of the statement in terms of the values of other statements. For example, the truth of the statement **Q1 > Q2+Q3** depends on the values of quantities **Q1**, **Q2** and **Q3**.

A statement can also be *assumed* to have a particular value, for instance, one can assume that **Q3** has the value 1 or that **Q1 > Q2** is true. We use the term *assumption* to indicate that the value of a statement is assumed to be true. For example, the assumption that block **A** is clear at time **t1** means that **Clear(A, t1)** is assumed to be true. A bug (inconsistency) arises when the desired (assumed) value of a statement conflicts with its predicted value obtained by evaluating the statement. For example, it is inconsistent to make the assumption that **Q1 > Q2** if **Q1** is already assumed to have the value 1.5 and **Q2** is assumed to be 2.0.

Events are represented as first-class objects in our world model. An event, also referred to as an *occurrence*, affects the state of the world by changing the values of attributes. An event is described by its type and parameter bindings. For example, the four statements:

Occurs(Deposition, Deposition1),
Parameter-of(Deposition1, Rock, SH),
Parameter-of(Deposition1, Dcomposition, Shale), and
Parameter-of(Deposition1, Dlevel, 100)

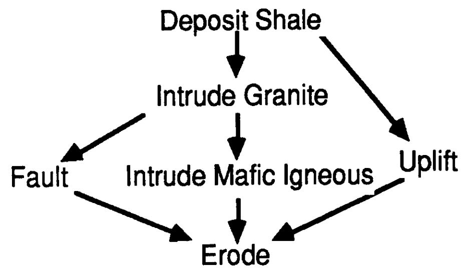
indicate that **Deposition1** is an event of type deposition[3] that creates a shale rock-unit named **SH** whose thickness is 100 (meters). For conciseness, this will often be written simply as:

Deposition1(SH, Shale, 100).

Deposition1 is constrained to occur during the interval from **Deposition1.start** to **Deposition1.end**. Ordering relations, such as the statement that **Deposition1** precedes event **Erosion1**, are written as **Deposition1.end < Erosion1.start**.

For the purpose of this report, an *hypothesis* is a set of events, represented by a collection of **Occurs** statements, **Parameter-of** statements and temporal relations between events. A *solution* to a problem is an hypothesis that can achieve the problem's goal state from its initial state. An hypothesis may specify only partially the parameter bindings and ordering of events. A partially ordered hypothesis is pictured graphically using arrows to represent the ordering of events, while totally ordered sequences are usually depicted with the events numbered (see Figure 9).

[3] To increase readability, the types of events are not always written in boldface.

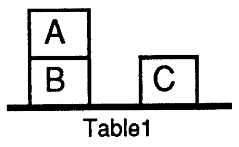


1. Deposit Shale
2. Intrude Granite into Shale
3. Uplift
4. Intrude Mafic Igneous through Granite and Shale
5. Fault across Shale and Granite
6. Erode Shale and Mafic Igneous

Figure 9. Partially and Totally Ordered Hypotheses

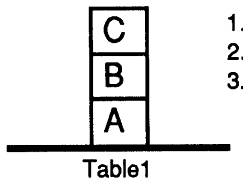
A *world state* (or just *state*) is a collection of statements that partially specifies the state of the world at some point in time or over an interval of time. Two important states for interpretation and planning problems are the *initial state* and *final* (or *goal*) *state*. Although many of the reasoning techniques used in GORDIUS do not require it, for simplicity this report restricts the initial and goal states to single points in time. The time points of the initial and final states are designated **Plan-start** and **Plan-end**, respectively. For example, Figure 10 illustrates both graphically and propositionally the initial and goal states for a blocks-world planning problem. In Section 7.5, we briefly discuss the use of GTD where the initial and goal states are not limited to single points in time, for instance, where the goal state is "robot is at MIT at **Plan-end** and robot is at concert at 8PM."

Initial State



1. On(A, B, Plan-start)
2. On(B, Table1, Plan-start)
3. On(C, Table1, Plan-start)
4. Clear(A, Plan-start)
5. Clear(C, Plan-start)

Goal State



1. On(A, Table1, Plan-end)
2. On(B, A, Plan-end)
3. On(C, B, Plan-end)

Figure 10. Initial and Goal States for a Blocks-World Planning Problem.

1.6 Outline

Chapters 2, 3 and 4 describe the details of the Generate, Test and Debug techniques, respectively. Each chapter is divided into three sections. The first sections describe GORDIUS' behavior in solving the example of Section 1.3 in more detail. The second sections of each chapter present details of the representations and reasoning methods used for each technique. The third sections discuss the strengths and weaknesses of each technique. In particular, Section 4.3 analyzes our theory of debugging plans and interpretations.

Chapter 5 presents the results of experiments using GORDIUS to solve problems in the geologic, blocks-world, Tower of Hanoi, and semiconductor fabrication domains. The relationship between the associational reasoning used by the generator and the causal reasoning used by the debugger is analyzed in Chapter 6. This chapter also describes the outline of an algorithm for deriving associational rules from the results of debugging problems, and it presents guidelines for other domains in which the GTD paradigm may be useful. Chapter 7 discusses related work and speculates on future directions for research in GTD-type paradigms that combine associational and causal reasoning.

2. Generate

The role of the generator is to hypothesize a set of events that can achieve the goal state of a problem. The generator constructs hypotheses using a *match* and *compose* algorithm and a library of *scenarios.* The scenarios associate patterns in the input with *local interpretations*, which are sets of events that can achieve the patterns.[1] The generator matches scenario patterns against the goal and initial states and composes together the local interpretations of matching scenarios to form a complete hypothesis. GORDIUS' generator algorithm is fairly domain-independent — by supplying different sets of scenarios, we have generated both geologic interpretations and blocks-world plans.

The generator starts by matching scenario patterns against the goal and initial states using a technique based on rete-nets. It then examines each unachieved goal proposition and finds those matching scenarios whose pattern contains the goal. For each such scenario, the generator composes its local interpretation with the current hypothesis, combining their events to create a new hypothesis. The new hypothesis is presumed to achieve all the goals of the scenario pattern plus the goals achieved by the original hypothesis.

A scenario provides only an heuristic indication, and not a guarantee, that the local interpretation actually formed the scenario pattern because the same pattern can often be formed by different combinations of events. For example, the pattern of a sedimentary and igneous rock-unit sharing a common boundary can be caused by a dike intruding through the sedimentary rock-unit, by a batholith intruding into the sedimentary rock-unit, or by the sedimentary rock-unit being deposited onto an existing igneous formation.

To determine efficiently which events actually formed a particular pattern, the generator uses scenarios that encapsulate patterns of interaction, that is, scenarios whose local interpretations are independent of other events in the hypothesis. Using independent scenarios enables the system to generate hypotheses by finding explanations for sets of goals separately and then combining the

[1] The term "local interpretation" is historical — scenarios can be used for either interpretation or planning. For interpretation, a local interpretation represents events that could have led to certain goals; for planning it represents events that should be done to achieve the goals.

explanations without the need for detailed checking for interactions between events.

Unfortunately, because of the large number of potential interactions between events in most domains, the best we can do is to create scenarios that nearly, but not totally, encapsulate interactions. Even so, the GTD paradigm presumes that scenarios can be composed independently, or with at most simple checks for inconsistencies. It is the role of the tester to determine if this presumption of composability does in fact hold, and it is the role of the debugger to repair hypotheses when the presumption does not hold.

The next section presents a concrete example of how GORDIUS generates a geologic interpretation. Section 2.2 describes the representations and reasoning techniques used by the generator in more detail. Section 2.3 analyzes how the associational reasoning used by the generator achieves efficiency by using scenarios that encapsulate common patterns of interaction.

2.1 Generation of an Initial Hypothesis

The input to the generator is a list of propositions describing the initial and goal states of the problem. For geologic interpretation, the initial state consists simply of the propositions that the surface of the Earth is bedrock, and that it lies below sea-level. The goal state describes the topological, geometrical, and geological properties of a vertical cross-section through the Earth.

For ease of use, the user describes the topological and geometrical properties of the goal diagram (Figure 11) using a representation based that concisely encodes the topological relationships between rock-units (faces) and boundaries (edges) [Baumgart]. GORDIUS translates this diagrammatic representation into a list of goal propositions that represents the properties that geologists find important in solving interpretation problems.

The important topological, geometrical, and geological properties are represented using the predicates **Abuts**, **Orientation** and **Composition**. For example, the section of the diagram with boundary **B5** between rock-units **SH1** and **SH2** translates to the seven goals:

Abuts(SH1, B5, Plan-end),[2]
Abuts(SH2, B5, Plan-end),
Orientation(B5, 100°, Plan-end),
Orientation(SH1, 0, Plan-end),
Orientation(SH2, 0, Plan-end),
Composition(SH1, Shale, Plan-end),
Composition(SH2, Shale, Plan-end).

For the diagram of Figure 11, there are a total of 28 **Abuts** goals (two for each boundary), 17 **Orientation** goals (one for each boundary plus one for each sedimentary rock-unit) and 7 **Composition** goals (one for each rock-unit). In addition, there is one goal stating that there are no other objects within the window of the goal diagram.

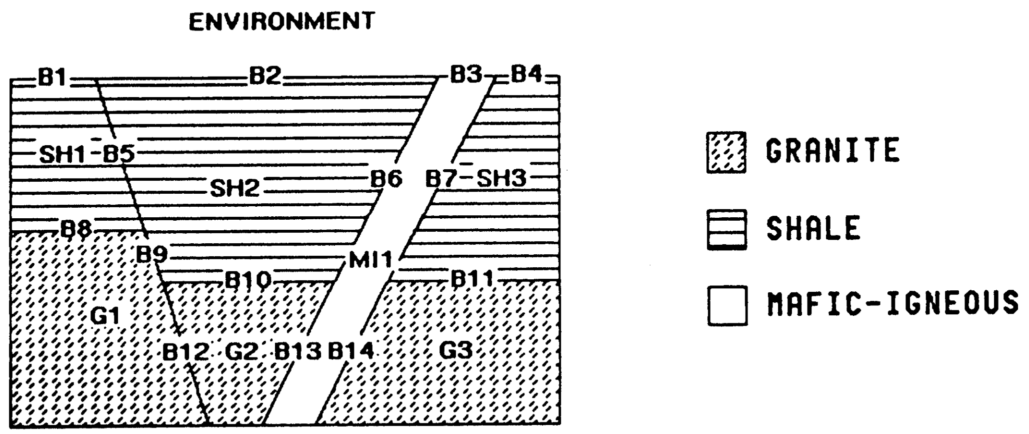


Figure 11. Geologic Interpretation Example

The generator currently uses 15 scenarios that we have found to be useful for the interpretation problems examined. Appendix A presents the scenario patterns and local interpretations that purport to explain how those patterns could have arisen (the representation language used is described in Section 2.2.1). In the example below, the generator needs only six of the scenarios to construct a solution to Figure 11.

The generator starts interpreting Figure 11 by determining why **MI1** abuts **B6**, which is the first in the list of goal propositions. One scenario that can interpret this goal is the "intrudes-through" scenario, which matches the section of the diagram where **MI1** is between **SH2** and **SH3**. The local interpretation of the "intrudes-through" scenario indicates that after some event created the

2 **Plan-end** is the time point associated with the goal diagram.

formation **ROCK1** (consisting of pieces **SH2** and **SH3**) a dike-intrusion pushed the mafic-igneous **DIKE1** (consisting of piece **MI1**) and the intrusional boundary **INTBOUND** (consisting of pieces **B6** and **B7**) through **ROCK1**, splitting it in two.

Another match of the "intrudes-through" scenario is used to interpret why **MI1** is between **G2** and **G3**. This scenario match indicates that **MI1** intruded through **ROCK2**, splitting it into **G2** and **G3**. Note that both matches of the "intrudes-through" scenario hypothesize the occurrence of a dike-intrusion event. The generator unifies the two events together to indicate they are actually the same event, since both interpret how the same piece of rock (**MI1**) was created. After composing the two "intrudes-through" local interpretations, the current hypothesis is:

$$\begin{tikzcd}
\text{Rock-Creation1(ROCK1)} \arrow[dr] & & \text{Rock-Creation2(ROCK2)} \arrow[dl] \\
& \text{Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)} &
\end{tikzcd}$$

where **SH2** and **SH3** are pieces split from **ROCK1**, **G1** and **G2** are pieces of **ROCK2**, **MI1** is a piece of **DIKE1**, and **B6**, **B7**, **B13** and **B14** are all pieces of the intrusional boundary **INTBOUND**.

Next, a match of the "simple-fault" scenario is used to interpret why **SH1** abuts **B5**. This scenario match indicates that boundaries **B5**, **B9** and **B12** are all pieces of **FAULT1**, a fault line that split the existing shale and granite formations. Since the "simple-fault" scenario interprets that **SH1** and **SH2** are pieces of the same formation, the generator infers by transitivity that **SH1** is also a piece of the **ROCK1** formation. Similarly, all three granite rock-units are inferred to be pieces of the **ROCK2** formation. The current hypothesis is now:

$$\begin{tikzcd}
\text{Rock-Creation1(ROCK1)} \arrow[d] \arrow[dr] & \text{Rock-Creation2(ROCK2)} \arrow[d] \arrow[dl] \\
\text{Faulting1(FAULT1)} & \text{Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)}
\end{tikzcd}$$

Next, the "erosion-ends-boundary" scenario is used to explain why **B1** abuts the **Environment**. The local interpretation indicates that **B1** and **B2** are pieces of an erosional boundary **EROBOUND** and that the level of erosion (**Elevel1**) was enough to affect rock-units **SH1** and **SH2** (pieces of **ROCK1**) and boundary **B5** (a piece of **FAULT1**). The local interpretation also hypothesizes that before the erosion, enough uplift occurred (**Uamount1**) to lift the surface of the Earth above sea-level. Note that to show off the power of the debugger in the example of Section 1.3, the "erosion-ends-boundary" scenario was modified by removing the uplift event. Here we use the complete "erosion-ends-boundary" scenario to show how the generator produces a correct solution without the need of the debugger.

At this point, the generator tries to explain why **B3** abuts the **Environment**. One scenario that matches this goal is "intrusion-to-surface," which hypothesizes that **MI1** intruded to the surface of the Earth, that **B3** is an intrusional boundary, and **B2** and **B4** are depositional boundaries. This interpretation, however, is inconsistent with the current hypothesis that **B2** is an erosional boundary. As a result, the "intrusion-to-surface" local interpretation is not composed with the current hypothesis.

Another scenario that can explain why **B3** abuts the **Environment** is the "erosion-ends-boundary" scenario. In this case, the scenario's local interpretation is consistent with the current hypothesis. A further application of the "erosion-ends-boundary" scenario is used to explain why **B4** abuts the **Environment**, producing the hypothesis:

$$\begin{tikzcd}
 & \text{Rock-Creation1(ROCK1)} \arrow[d] \arrow[dr] & \text{Rock-Creation2(ROCK2)} \arrow[d] \arrow[dl] \\
\text{Uplift1(Uamount1)} \arrow[dr] & \text{Faulting1(FAULT1)} \arrow[d] & \text{Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)} \arrow[dl] \\
 & \text{Erosion1(EROBOUND, Elevel1)} &
\end{tikzcd}$$

Three matches of the "sedimentary-no-tilt" scenario are now used to determine why **SH1**, **SH2** and **SH3** have shale composition and orientations of zero degrees. The local interpretations indicate that the shale rock-units (all pieces of **ROCK1**) were all created by a deposition event (**Deposition1**). The generator unifies together the **Deposition1** and **Rock-Creation1** events since they both purport to create the same formation. In the new hypothesis, **Rock-Creation1** is replaced by **Deposition1**, the more specialized of the two events.

Similarly, three matches of the "igneous-under-sedimentary" scenario are used to determine why **G1**, **G2** and **G3** are granite. As above, the generator uses these scenarios to specialize **Rock-Creation2** to be a batholithic-intrusion event. In addition, the local interpretation indicates that the intrusion of granite occurred after the deposition of shale. Note that the "sedimentary-over-igneous" scenario also interprets why **G1, G2** and **G3** are granite, but the generator prefers the "igneous-under-sedimentary" scenario since it produces an hypothesis with fewer events:

$$\begin{tikzcd}
 & \text{Deposition1(ROCK1, Shale)} \arrow[d] & \\
 & \text{Batholithic-Intrusion2(ROCK2, Granite)} \arrow[d] \arrow[dr] & \\
\text{Uplift1(Uamount1)} \arrow[dr] & \text{Faulting1(FAULT1)} \arrow[d] & \text{Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)} \arrow[dl] \\
 & \text{Erosion1(EROBOUND, Elevel1)} &
\end{tikzcd}$$

At this point, all the goals have been interpreted. The generator now tries to determine whether any of the scenarios can be used to order the currently unordered events. Although no matching scenarios constrain the order of the faulting and dike-intrusion events, the "eroded-sedimentary" scenario can be used to order the uplift and deposition event. This scenario indicates that uplift must occur between deposition and erosion, since deposition occurs under water and erosion occurs in the air. Taking this scenario into account, the initial hypothesis proposed by the generator is:

$$\begin{tikzcd}
 & \text{Deposition1(ROCK1, Shale)} \arrow[d] \arrow[ddl] & \\
 & \text{Batholithic-Intrusion2(ROCK2, Granite)} \arrow[d] \arrow[dr] & \\
\text{Uplift1(Uamount1)} \arrow[dr] & \text{Faulting1(FAULT1)} \arrow[d] & \text{Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)} \arrow[dl] \\
 & \text{Erosion1(EROBOUND, Elevel1)} &
\end{tikzcd}$$

Testing this hypothesis, as described in Section 3.1, verifies that it is a correct solution to the problem in Figure 11.

2.2 The Generator

This section contains a detailed description of the match and compose algorithm used by our generator. The next two sections describe the representations and reasoning techniques used to match scenario patterns and compose local interpretations. Section 2.2.3 discusses how the best-first search for a solution is controlled — an important consideration in domains, such as geologic interpretation, where the search space of hypotheses is often very large.

2.2.1 Matching Scenario Patterns

The pattern of a scenario represents the conditions under which the scenario may be applicable. By finding which scenario patterns match against the input, the generator determines which associated local interpretations could have occurred to achieve the goals of the problem.

A scenario pattern is represented as a list of propositions containing free variables. For example, **Abuts(?r1, ?eb1, Plan-end)** denotes all sets of rock-units and boundaries that touch one another at time **Plan-end** (a term starting with **?** denotes a free variable). In matching scenario patterns, GORDIUS determines bindings for the variables, which in turn provides bindings for instantiating the local interpretation associated with the pattern.

The propositions in a scenario pattern are divided into a *goal part* and an *initial part,* which match against the goal state (time **Plan-end**) and initial state (time **Plan-start**), respectively (see Figures 12 and 13).[3] The two parts of a scenario pattern represent different aspects of the scenario's applicability. The goal part, which describes observable effects that will hold after the scenario's local interpretation occurs, provides a way to recognize which events occurred (or, in planning domains, which events should occur to achieve the goals). For example, Figure 12 is a pattern that typically arises when erosion occurs. In Figure 13, the goal part indicates that the scenario is applicable if one has the goal of stacking three (or more) blocks.

The initial part of a scenario pattern, which represents necessary conditions for the local interpretation to occur, is used to determine the context under which the scenario can be used. For example, the initial part of Figure 13 indicates that in order to stack three blocks simply by putting block **B** on **C** and then **A** on **B**, all three blocks initially need to be clear on top. For our geologic scenarios, it is not necessary to include an initial part to limit the applicability of scenarios since in our interpretation problems the initial state is always the same.

[3] Some geologic scenarios include timeless propositions, such as **?theta $\neq$ 0**, that for convenience are presumed to belong to the goal part.

Pattern

Goal Part: Abuts(?r1, ?b1, Plan-end)
Abuts(?r2, ?b2, Plan-end)
Abuts(Environment, ?b1, Plan-end)
Abuts(Environment, ?b2, Plan-end)
Orientation(?b1, ?theta, Plan-end)
Orientation(?b2, ?theta, Plan-end)
Abuts(?r1, ?b3, Plan-end)
Abuts(?r1, ?b3, Plan-end)
Connected(?b1, ?b3, Plan-end)
Connected(?b2, ?b3, Plan-end)
Connected(?b1, ?b2, Plan-end)

Schematic of Pattern

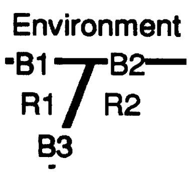
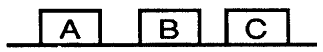


Figure 12. The "Erosion-Ends-Boundary" Scenario Pattern

Initial Part: Clear(?a, Plan-start)
Clear(?b, Plan-start)
Clear(?c, Plan-start)



Goal Part: On(?a, ?b, Plan-end)
On(?b, ?c, Plan-end)

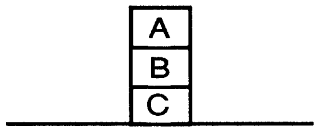
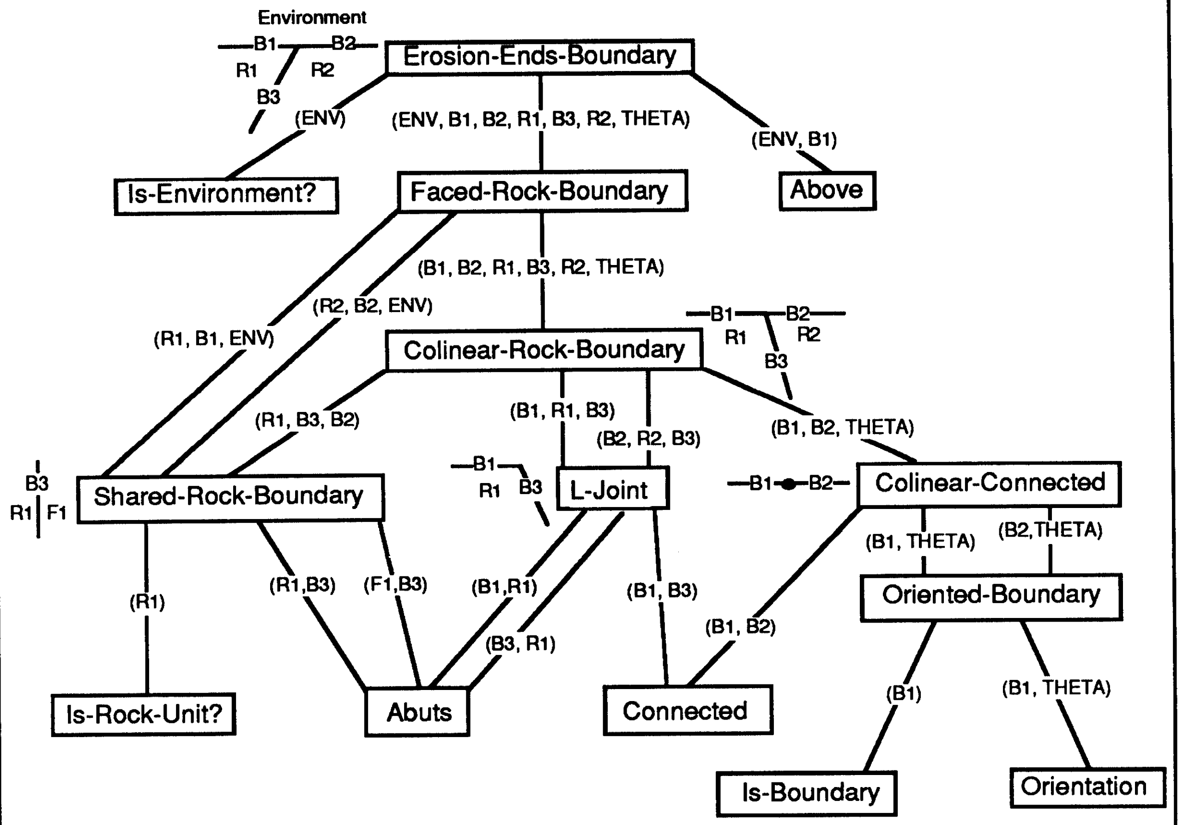


Figure 13. Blocks-world Scenario Pattern For Stacking Three Blocks (from Appendix D)

The generator matches the propositions of a scenario pattern against the propositions describing the goal and initial states and constructs sets of binding lists for the free variables in the pattern. Matching is implemented using rete-networks [Forgy], a technique for efficiently matching patterns that share a large number of common sub-patterns, as is typical of our geologic and blocks-world domains.

The rete-network represents a scenario pattern as a binary tree where the leaves, called *match nodes*, represent propositions and the non-leaf nodes, called *merge nodes*, represent conjunctions of propositions. Scenarios with similar patterns share the same match and merge nodes up to the point where their patterns diverge. To further increase the amount of sharing possible, and hence the efficiency of matching scenarios, our rete-networks can incorporate sub-trees, which we call *defined predicates*, that behave like match nodes. For example, a common pattern in the geologic domain is where a rock-unit shares a common boundary with another geologic object (**R1 | F1**). This "shared-rock-boundary" defined predicate appears three times in Figure 14, which shows the rete-net representation of the "erosion-ends-boundary" scenario of Figure 12.



Arcs are labeled with the parameter binding list passed between the rete nodes.
Schematics are shown for some nodes, indicating the pattern they match.
All nodes have a temporal argument, not shown here for simplicity.

Figure 14. Rete-Network Representation of the "Erosion-Ends-Boundary" Scenario Pattern

In a rete-network, a match node collects variable bindings for each matching instance of its associated proposition and passes the binding lists up to its parent merge nodes (the arcs in Figure 14 are labelled with the variables passed between nodes). A merge node receives binding lists from its two children nodes and, after checking for consistency, combines the binding lists and passes them up to its parent nodes. A complete scenario pattern matches if its top-most merge node receives a non-empty set of binding lists.

Our pattern matcher differs from traditional rete-networks in two important respects — 1) our matcher is goal-driven rather than data-driven and 2) our matcher extends the method used by merge nodes to combine variable bindings received from children nodes.

Traditional rete-networks are data-driven, in that match nodes are activated whenever a new instance of their proposition is found and merge nodes are activated whenever they receive bindings from their children nodes. In contrast, our pattern matcher is used in a goal-driven fashion. A merge node is activated when it receives a partial (possibly empty) list of bindings from one of its parent nodes. The merge node passes the bindings down to its left child, activating it, then in turn passes each of the binding lists received from its left child down to its right child, and finally returns the resultant set of binding lists back to the node that activated it.

For our domains, the goal driven pattern matcher is more efficient than a data-driven one would be. This is due mainly to the combinatorics of matching large, under-constrained data sets, as is typical in the geologic domain. For example, to find all matches of the "shared-rock-boundary" predicate **(R1|F1)**, a data driven pattern matcher would have to examine all pairs of rock-units and geologic objects — an $O(N^2)$ operation. In the actual goal-driven matcher, often only $O(N)$ pairs need to examined since the initial parts of the pattern typically constrain the bindings of one of the objects.

The other difference from traditional rete-networks is in the method used by merge nodes to combine variable bindings received from their children nodes. In traditional rete-networks, binding lists can be combined as long as they do not bind the same variable differently. We have added two extensions that have proven useful for matching scenario patterns. First, pairs of variables (**?x**, **?y**) can be declared to be symmetric. The pattern matcher ensures that if a binding list is constructed with **?x** bound to **A** and **?y** bound to **B** then no binding list will be constructed with **?x** bound to **B** and **?y** to **A**. Symmetries are used to prevent the matcher from finding redundant matches. For example, the "erosion-ends-boundary" pattern (Figure 12) is symmetric in **?r1** and **?r2** since the interpretation of the pattern does not depend on whether the rock-unit bound to **?r1** is to the left or right of **?b3** in the goal diagram.

The second extension for combining variable bindings is that the pattern matcher ensures that no binding appears more than once in a binding list, under the presumption that all variables are unique. This presumption of uniqueness usually corresponds to our intuitive notions of how patterns should be matched. For example, the "shared-rock-boundary" pattern:

Abuts(?r1, ?b, Plan-end) and Abuts(?f1, ?b, Plan-end)

is meant to represent that **?r1** and **?f1** share a common boundary in the goal state. We want **?r1** and **?f1** to match uniquely since we usually do not want to infer that a rock-unit shares a common boundary with itself. In the future, however, we plan to add the ability to turn off this feature for selected pairs of variables, since at times assuming uniqueness of variables is too restrictive.

2.2.2 Composing Local Interpretations

This section describes how local interpretations are represented and composed. The local interpretation of a scenario is a set of events that can plausibly achieve the scenario's pattern. Each matching scenario can thus explain how a set of goals arose. To explain how *all* the goals of a problem arose, the generator composes together local interpretations from different matching scenarios.

The composition step instantiates a local interpretation, using variable bindings derived from matching the scenario pattern. After the local interpretation is instantiated, a new hypothesis is generated simply by adding the events of the local interpretation to those of the current hypothesis. For the most part, the events are combined without checking for interactions. However, some inexpensive checks are made to detect obvious inconsistencies, such as cycles in the ordering of events (see Section 2.2.3).

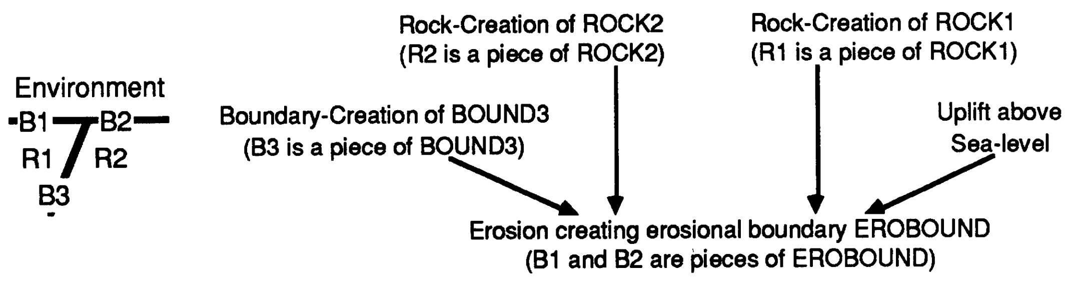
Local interpretations are represented using four different types of constraints:

1. Occurrence constraints describe which events occur. An **Occurs** statement includes the type and name of the event. For example, the local interpretation of the "erosion-ends-boundary" scenario (Figure 15) states that the pattern arises from the creation and subsequent erosion of two rock-units and one boundary.[4]

2. Parameter-binding constraints describe the objects that take part in the events. For example, the "erosion-ends-boundary" scenario indicates that **Erosion1** creates **Erobound**, the

[4] The rock-creation and boundary-creation events do not necessarily represent different events. As described later in this section, the generator may unify them together if they are found to refer to the same event.

<u>Schematic Representation of Scenario Pattern and Local Interpretation</u>



```
1. Occurs(Rock-Creation, ?rock-creation1)
   Occurs(Rock-Creation, ?rock-creation2)
   Occurs(Boundary-Creation, ?boundary-creation3)
   Occurs(Uplift, ?uplift1)
   Occurs(Erosion, ?erosion1)

2. Parameter-of(?rock-creation1, Rock, ?rock1)
   Parameter-of(?rock-creation2, Rock, ?rock2)
   Parameter-of(?boundary-creation3, Boundary, ?bound3)
   Parameter-of(?erosion1, Boundary, ?erobound)
   Parameter-of(?erosion1, Elevel, ?elevel1)
   Parameter-of(?uplift1, Uamount, ?uamount)
   ?uamount≥ Sea-Level – Surface.top.height@?uplift1.Start ;; Enough uplift to raise the
                                                              surface of the Earth above sea-level
   ?elevel1≤ ?rock1.top.height@?erosion1.start          ;; Enough erosion to affect rock1
   ?elevel1≤ ?rock2.top.height@?erosion1.start          ;; Enough erosion to affect rock2

3. ?rock-creation1.end ≤ ?erosion1.start
   ?rock-creation2.end ≤ ?erosion1.start
   ?boundary-creation3.end ≤ ?erosion1.start
   ?uplift1.end ≤ ?erosion1.start

4. Piece-of(?r1, ?rock1)
   Piece-of(?r2, ?rock2)
   Piece-of(?b1, ?erobound)
   Piece-of(?b2, ?erobound)
   Piece-of(?b3, ?bound3)
```

Figure 15. The "Erosion-Ends-Boundary" Local Interpretation (from Appendix A)

erosional boundary between the environment and the eroded rock-units. The local interpretation can also include arithmetic constraints on the values of parameters. For example, Figure 15 indicates that enough uplift (**Uamount**) occurs to raise the surface of the Earth above sea-level.

3. Temporal-ordering constraints describe the order in which events occur. Although our temporal representations can

support a full range of orderings [Simmons, 86], in practice we have found it sufficient to use only one type of ordering for the domains explored — one event precedes another, that is, the end point of one event is less than or equal to the start point of the other event.

4. Piece-of constraints describe piece/whole relationships between objects. **Piece-of(P,W)** means that the matter forming object **P** is a subset of the matter forming the larger object **W**. **Piece-of** constraints enable GORDIUS to aggregate physically discrete objects by their common origins. For example, the "erosion-ends-boundary" scenario states that the boundaries abutting the environment are pieces of the same erosional boundary. By matching the "erosion-ends-boundary" scenario against boundaries **B1-B4** in Figure 11 and using the transitivity of the **Piece-of** relation, GORDIUS can infer that **B1-B4** are all pieces of the same boundary and thus were formed by the same erosion event.

The basic step in composing a local interpretation with the current hypothesis is to instantiate the local interpretation by determining bindings for its variables. The generator determines bindings in three stages: 1) the binding list constructed in matching the scenario pattern provides an initial set of bindings; 2) additional bindings are dictated by the current hypothesis; 3) remaining free variables are bound to unique names chosen arbitrarily by GORDIUS.

The mechanisms used in the first and third stages should be evident; the rest of this section describes the second stage. We have identified several ways in which the current hypothesis dictates the choice of variable bindings for a local interpretation. Each of these methods tries to choose bindings by unifying events in the local interpretation with events in the current hypothesis.

Before describing the methods themselves, we introduce two concepts used by them — that of *potentially* and *necessarily unifiable* events. Two events are potentially unifiable if the following three conditions hold:

1. The events are temporally unordered with respect to one another.
2. The events have compatible types. Compatible means the type of one event is the same as, or more specific than, the type of the other. For example, **Deposition** and **Rock-Creation** are

compatible types, but **Batholithic-Intrusion** and **Dike-Intrusion** are not.

3. The parameter values of the two events cannot be shown to be different. For example, deposition of shale and deposition of sandstone are not potentially unifiable since shale and sandstone are different materials; on the other hand, uplift of more than 500 meters and uplift of less than 1000 meters *are* potentially unifiable, since there are values for the uplift parameter that satisfy both constraints.

Two events are *necessarily unifiable* if they are potentially unifiable and purport to create at least some of the same objects. For example, a rock-creation event that creates rock-unit **ROCK1** and a deposition event that creates **ROCK1** and boundary **BOUND1** are necessarily unifiable. Such events are necessarily the same event because an object in the physical world can be created in only one way and by only one event.

Two unifiable events are actually unified by combining their temporal-ordering and parameter-binding constraints. The name and type of a unified event is chosen to be that of the more specific of the two event types (e.g., **Deposition** is more specific than **Rock-Creation**). Similarly, the name of a unified parameter binding is the object with the more specific type.

Getting back to choosing variable bindings, the generator uses three methods for choosing bindings based on the current hypothesis, each based on simple premises about the physical world: 1) an object that is a piece of some aggregate in the current hypothesis must be a piece of the same aggregate in the local interpretation, under the premise that an object can be a piece of only one aggregate; 2) two events that purport to create the same object are hypothesized to be the same event, under the premise that an object can be created by only one event; and 3) two events that have the same effects may be the same event, under the premise that events have multiple effects and can achieve multiple goals. The actual methods used are described in more detail below.

The first method looks for **Piece-of** constraints in the local interpretation that match existing **Piece-of** constraints. For each constraint in the local interpretation of the form **Piece-Of(?p, ?w)**, where **?p** is already bound to some object **P1**, the generator looks for any constraints in the current hypothesis of the form

Piece-of(P1, W1). If **?w** is currently unbound, **?w** is bound to **W1**. If **?w** is already bound to a different object **W2**, the generator determines whether the event that created **W2** is necessarily unifiable with the event that created **W1**. If so, **W1** and **W2** are constrained to be the same object, and the events that created them are unified; otherwise the local interpretation is inconsistent with the current hypothesis and, therefore, cannot be composed with it.

This method for choosing bindings is based on the premise that an object can be a piece of only one aggregate object. Although this premise holds for geologic interpretation, we do have sufficient experience with other domains that support piece/whole relationships to comment on the general applicability of this premise or, for that matter, the applicability of our representation for pieces of objects.

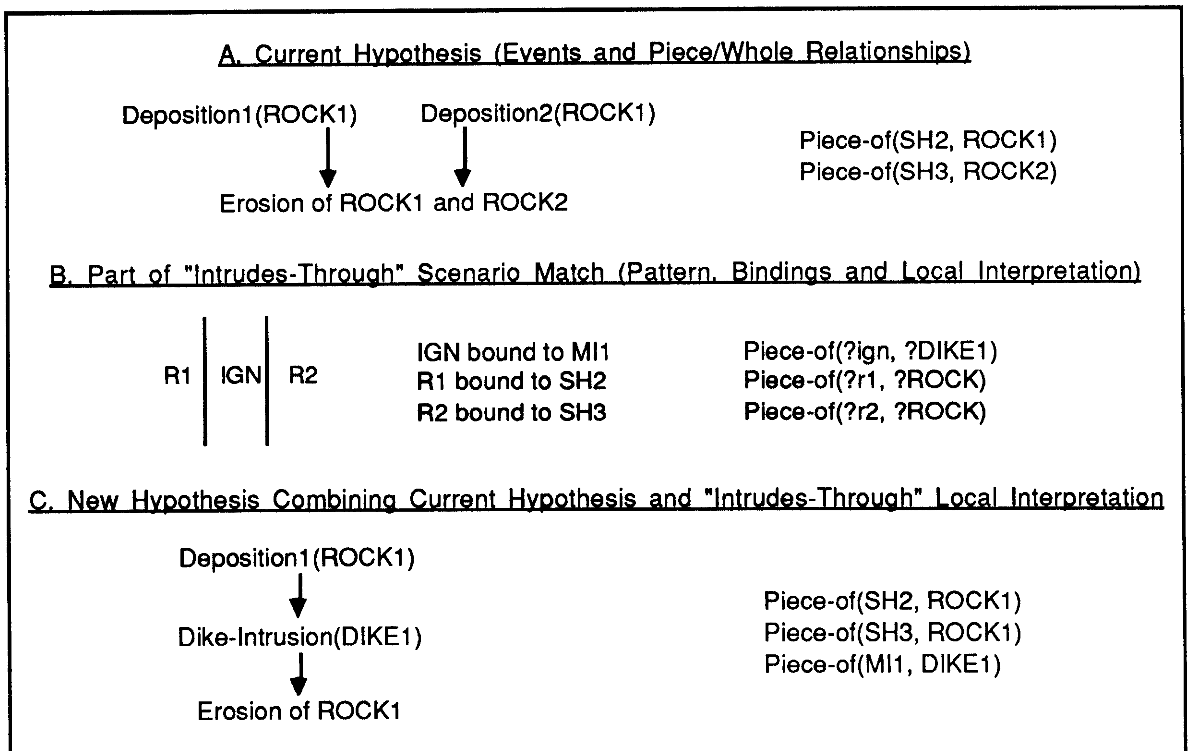


Figure 16. The Generator Unifies Objects and the Events that Created Them.

Figure 16 illustrates this method for choosing bindings. The current hypothesis (Figure 16a) has erosion preceded by two unordered deposition events, where **SH2** and **SH3** are pieces of the deposited formations **Rock1** and **Rock2**, respectively. The generator is composing the local interpretation of the "intrudes-through" scenario, where the current bindings indicate that an igneous rock-unit **MI1** is between **SH2** and **SH3** (Figure 16b).

The generator finds that the constraint **Piece-Of(?r1, ?rock)**, where **?r1** is bound to **SH2**, matches **Piece-of(SH2, ROCK1)** in the current hypothesis, so **?rock** is bound to **ROCK1**. Similarly, since the constraint **Piece-Of(?r2, ?rock)** matches **Piece-of(SH3, ROCK2)**, the generator determines that **?rock** must also be bound to **ROCK2**. Since the deposition event that created **ROCK1** and the deposition event that created **ROCK2** are necessarily unifiable (they are unordered, of the same type, etc.), the generator presumes that **ROCK1** and **ROCK2** are actually the same object and unifies the two deposition events into one. Figure 16c illustrates the resulting hypothesis, including the newly added dike-intrusion event.

The second method for using the current hypothesis to choose variable bindings makes use of the premise that an object can be created by only one event. For each event **E1** in the local interpretation, the generator finds the events in the current hypothesis which create an object that is also purported to be created by **E1**. If there are such events, the generator determines whether each of them are necessarily unifiable with **E1** and with one another. If some are not unifiable, which could happen if the events do not have compatible types or they are already ordered, then the local interpretation is inconsistent with the current hypothesis and is not pursued further. Otherwise, the events are unified and the name and parameter bindings of the unified events are used to constrain the appropriate free variables in the local interpretation.

The third method chooses bindings for a local interpretation by trying to unify events in the local interpretation with potentially unifiable events in the current hypothesis. In effect, the generator tries to achieve the desired effects of the local interpretation by "reusing" existing events in the current hypothesis.[5] This method is based on the premise that events have multiple effects and can serve multiple purposes, for instance, a **Puton** event can be used both for stacking and for clearing blocks.

The generator unifies each event in the local interpretation with any potentially unifiable events. If an event potentially unifies with

[5] This same effect is achieved in planners such as NOAH [Sacerdoti] and SIPE [Wilkins] by introducing "phantom" nodes into the plan network.

more than one other event, the generator creates separate hypotheses to reflect the different unification choices. For example, suppose the generator has constructed the current hypothesis in Figure 17c to achieve the first two goals in Figure 17b (the first goal is achieved by putting **A** somewhere, then putting **B** on **C**; the second goal is achieved by putting something on **B**; **Puton2** and **Puton3** are linearized for simplicity). One of our blocks-world scenarios (see Appendix D) suggests that the third goal **On(A, B, Plan-end)** can be achieved using the event **Puton4(A, B)**. The generator reflects the fact that **Puton4** potentially unifies with both **Puton1** and **Puton3** by creating two new hypotheses:

1. Puton1(A, B)	**1. Puton1(Source=A)**
2. Puton2(B, C)	**2. Puton2(B, C)**
3. Puton3(Dest=B)	**3. Puton3(A, B)**

Note that while the second hypothesis is a solution, the first is incorrect since putting **A** on **B** interferes with the precondition of **Puton2** that **B** be clear. The question of which hypothesis to pursue further is the subject of the next section.

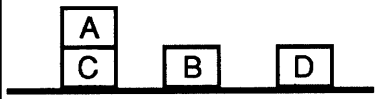
A. Initial State	B. Goal State	C. Current Hypothesis
	1. On(B, C, Plan-end) 2. Covered(B, Plan-end) 3. On(A, B, Plan-end)	1. Puton1(source=A) 2. Puton2(B, C) 3. Puton3(dest=B)

Figure 17. Composing Potentially Unifiable Events

2.2.3 Controlling the Generator

This section describes how the generator incorporates the techniques for matching scenarios and composing local interpretations into the overall search for an hypothesis that achieves all the goals of the problem. We describe how the search strategy used decides which scenarios to choose for matching, which hypotheses to pursue further, and what to do when an inconsistency is found in composing local interpretations.

GORDIUS begins with the null hypothesis (i.e., no events occur) and a list of goal propositions. For the blocks-world and Tower of Hanoi domains, the goal propositions are input manually. For the geologic domain, the system translates the goal diagram into a list of goal propositions that encode information about the orientation, composition, and topological adjacency of objects.

The basic action of the generator is to pick an unachieved goal and find the scenarios that explain how the goal can be achieved. The generator constructs new hypotheses by composing the local interpretations of matching scenarios with the current hypothesis, continuing until all the goals of the problem are hypothesized to be achieved.

The search space of the generator is rather large since, individually, each goal can typically be achieved in many ways. Our generator tries to minimize the search using heuristics. [Barr] identifies three decision points in search that can benefit from the use of heuristic information — 1) which hypothesis to pursue, 2) how to expand a hypothesis, and 3) how to handle dead ends. Our generator employs simple yet effective heuristics for making each of these decisions.

Our generator uses best-first search to decide which hypotheses to pursue. In best-first search, the hypothesis to pursue next is the one on the fringe of the search with the best value of the distance metric. The generator's metric relies primarily on how close the hypothesis is to solving the problem, which is estimated by counting the number of remaining unachieved goals. The metric's secondary component, used to break ties, is the cost of the hypothesis. Currently, we take the simple approach that each event has unit cost, so the cost is simply the number of events in the hypothesis. This distance metric is in keeping with the philosophy of the generator — it is inexpensive to compute and, although fairly rough, has proven adequate in practice.

Our choice of distance metric has several interesting consequences on the overall search strategy. One consequence is that the search tends to follow one path instead of jumping around, as often occurs in best-first search. This is because our primary measure of goodness is closeness to the goal state and expanding an hypothesis always decreases the number of unachieved goals (see below). Another consequence of the metric is that the generator tends to pursue hypotheses constructed using scenarios with more specific patterns. The generator tends to converge faster towards a solution using more specific scenarios since they explain more goals. This effect is only approximate, however, since some goals of a large scenario pattern may already be achieved, so a scenario with a smaller pattern may explain more of the currently unachieved goals.

The generator chooses the best hypothesis and expands it by selecting a set of matching scenarios and composing them with the current hypothesis. The generator selects matching scenarios by choosing the first remaining unachieved goal as the *focus item* and finding all matching scenarios where one of the items in the scenario pattern matches the focus item.

For each matching scenario, the generator tries to construct a new hypothesis by composing the scenario's local interpretation with the current hypothesis. Although, for the most part, the generator presumes that a local interpretation and current hypothesis can be composed, the generator employs several computationally efficient methods for detecting a limited class of inconsistencies. The generator determines that a composition is inconsistent if any of the following four situations arise: 1) different (non-unifiable) events create the same object, 2) an object has incompatible types, 3) two events are temporally inconsistent, and 4) an object is a piece of more than one aggregate object. For example, it is inconsistent for a local interpretation to assume that one event occurs before another event if the current hypothesis assumes that they occur in the opposite order.

If no inconsistency is found, a new hypothesis is generated by adding the constraints of the local interpretation to those of the current hypothesis. The generator records that the new hypothesis achieves all the goals that match an item in the scenario's pattern, plus all goals already achieved by the current hypothesis. The new hypothesis is then added to the set of unexplored hypotheses and the generator continues by choosing the best hypothesis from that set.

It is conceivable that the generator will find no matching scenarios whose pattern includes the focus item. This situation may arise if the author of the scenario library forgot to include a scenario that could explain the focus item. In this case, the generator simply chooses the next unachieved goal as the focus item and leaves it up to the debugger to determine how to achieve the unmatched focus item.

"Dead ends" are handled very simply by the generator. A dead end occurs when an inconsistency is found between a local interpretation and the current hypothesis. In such cases, no updated hypothesis is generated using that interpretation. If all matching scenarios are inconsistent with the current hypothesis, the current hypothesis is abandoned. A more sophisticated strategy would involve determining which scenarios used in generating the current hypothesis contribute to the inconsistency and creating an hypothesis that does not include those scenarios. This strategy, which is akin to dependency-directed backtracking, has not been implemented for our generator, since we want to keep it simple and inexpensive.

GORDIUS continues generating hypotheses until it finds one that achieves all the goals (more precisely, one that achieves all goals that are explainable using the current scenario library since, as indicated above, some goals may not match any existing scenario). The search is guaranteed to terminate because at each step the generator examines one or more focus items, monotonically decreasing the number of remaining goals that need to be examined while adding only a finite number of new hypotheses.

Before sending the hypothesis to the tester, the generator tries to constrain currently unordered events by finding scenarios whose local interpretations order those events. For example, the "erosion-ends-boundary" scenario states that uplift occurs before erosion, but does not order uplift with respect to the events that create the rock-units that get eroded. This is a reasonable interpretation since uplift does not have to follow the rock-creation events if they create igneous rocks. If one of the rock-units is sedimentary, however, the "eroded-sedimentary" scenario can be used to determine that uplift must follow deposition of the sedimentary rock in order to raise the surface enough for erosion to occur.

The mechanism to constrain the order of events uses many of the techniques described in this chapter for achieving goals. For each pair of unordered events, the generator looks for matching scenarios whose local interpretations include events that are ordered with respect to each other and are potentially unifiable with the pair of unordered events. For each such matching scenario whose local interpretation is consistent with the current hypothesis, a new hypothesis is produced by composing the local interpretation with the current hypothesis. After all pairs of unordered events have been focused on, the best hypothesis is sent to the tester.

In some cases, trying to constrain unordered events can have a major impact on the hypothesis. For example, Figure 18b shows the partially ordered hypothesis generated by GORDIUS to account for all the goals of Figure 18a. The hypothesis contains two unordered fault events — one whose fault line includes pieces **FB1, FB2** and **FB3**, the other including pieces **FB4, FB5** and **FB6**. In trying to order the events, GORDIUS matches the "igneous-cuts-boundary" scenario (-B1-| IGN |-B2-) against the part of the diagram where **DIKE1** crosses **FB3** and **FB4**. The local interpretation of this scenario indicates that **Dike-Intrusion1** split an existing boundary into two pieces, **FB3** and **FB4**. Since **FB3** and **FB4** are discovered to be pieces of the same boundary, the generator unifies the two fault events that created those boundaries into a single event. Thus, as a result of trying to order events, the generator infers that there is actually only one fault, cut by **DIKE1**.

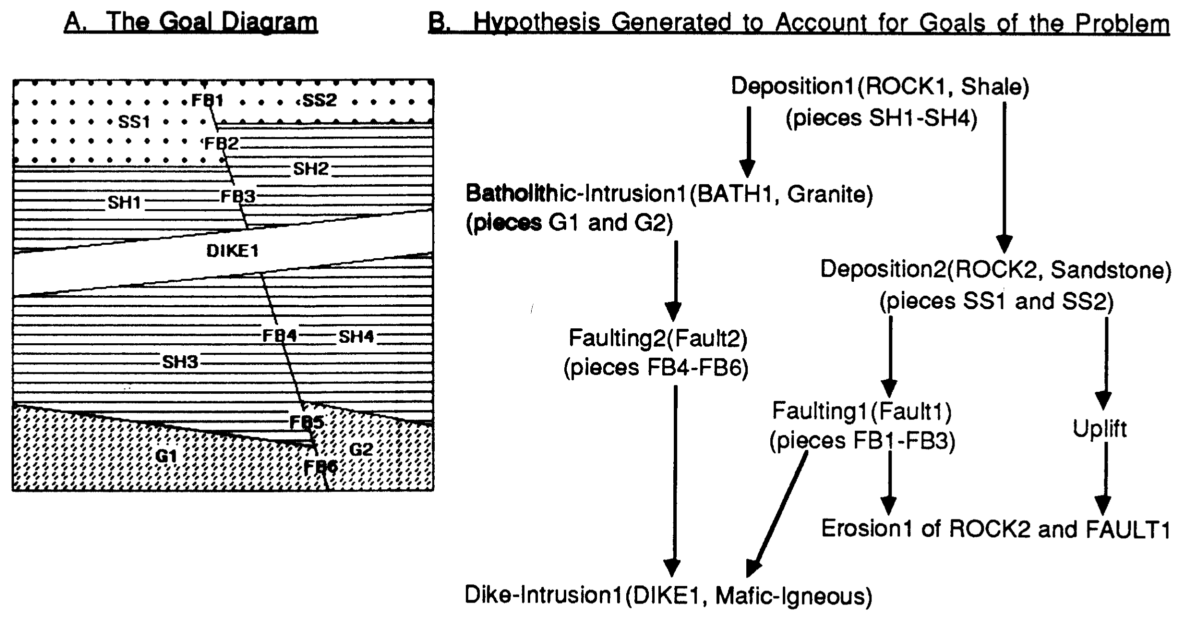

A. The Goal Diagram
B. Hypothesis Generated to Account for Goals of the Problem
SS1
FB1
SS2
FB2
SH2
SH1
FB3
DIKE1
FB4
SH4
SH3
FB5
G1
G2
FB6
Deposition1(ROCK1, Shale)
(pieces SH1-SH4)
Batholithic-Intrusion1(BATH1, Granite)
(pieces G1 and G2)
Deposition2(ROCK2, Sandstone)
(pieces SS1 and SS2)
Faulting2(Fault2)
(pieces FB4-FB6)
Faulting1(Fault1)
(pieces FB1-FB3)
Uplift
Erosion1 of ROCK2 and FAULT1
Dike-Intrusion1(DIKE1, Mafic-Igneous)

Figure 18. Unifying Events While Trying to Order Them.

2.3 Encapsulating Common Patterns of Interaction

The match and compose technique used by our generator provides a very efficient means of constructing hypotheses. The technique decomposes a problem into sub-problems based on the scenario patterns, solves the sub-problems independently using the local interpretations, and then combines the partial solutions to achieve all the goals of the problem.

An efficient generator is not very useful, however, unless it usually constructs hypotheses that are correct or nearly so. We argue in this section that the correctness of generated hypotheses depends on the degree to which the scenarios *encapsulate common patterns of interaction.*

A scenario encapsulates interactions if the events in its local interpretation are sufficient to explain how the scenario's pattern could be achieved and if no other events can interfere with the achievement of that pattern. Scenarios that totally encapsulate interactions are independent of one another, so complete and correct hypotheses can be formed simply by independently composing their local interpretations. Unfortunately, totally encapsulating interactions is not feasible in many domains, due to the large number of potential interactions between events. The GTD approach is to supply the generator with a small library of scenarios that nearly encapsulate *common* patterns of interaction and have the generator presume that no other interactions happen. If unanticipated interactions do arise, they will be detected by the tester and repaired by the debugger.

In analyzing our library of geologic and blocks-world scenarios, we have found that they encapsulate two types of interaction — interaction within a single event, and interaction between events. Interaction within a single event happens when effects are *coupled*, that is, when one effect does not (or is unlikely to) happen without the other happening as well. For example, a dike intrusion through a formation splits the formation, creating two new rock-units. These effects are coupled since the intrusion cannot create one of the new rock-units without creating the other as well.

Interaction between events happens in two ways: 1) one event affects a precondition (or change condition) of another event, and 2) two events affect the same attribute, for instance, two tilt events interact since they both act to increase the orientation of existing rock-units. We can further classify interactions between events as either cooperating or interfering. An example of cooperative interaction is where uplift and erosion cooperate in eroding a sedimentary formation, since uplift acts to achieve the preconditions of erosion. An example of interference is where putting block **A** on **B** interferes with subsequently putting **B** on **C**, since the preconditions for the latter action no longer hold.

The main concern in creating a scenario is how much of the coupled, cooperative, and interference interactions to encapsulate, which in turn determines the size and scope of the scenario patterns. In general, deciding how much interaction to encapsulate involves a tradeoff between generality and independence of the scenarios. If the patterns are too large, the scenarios tend to be overly specific and a large number are required to ensure adequate coverage of the domain. For example, a scenario whose pattern is the complete goal state of a problem and whose local interpretation is the complete solution is independent, but it is useful only for solving that particular problem.

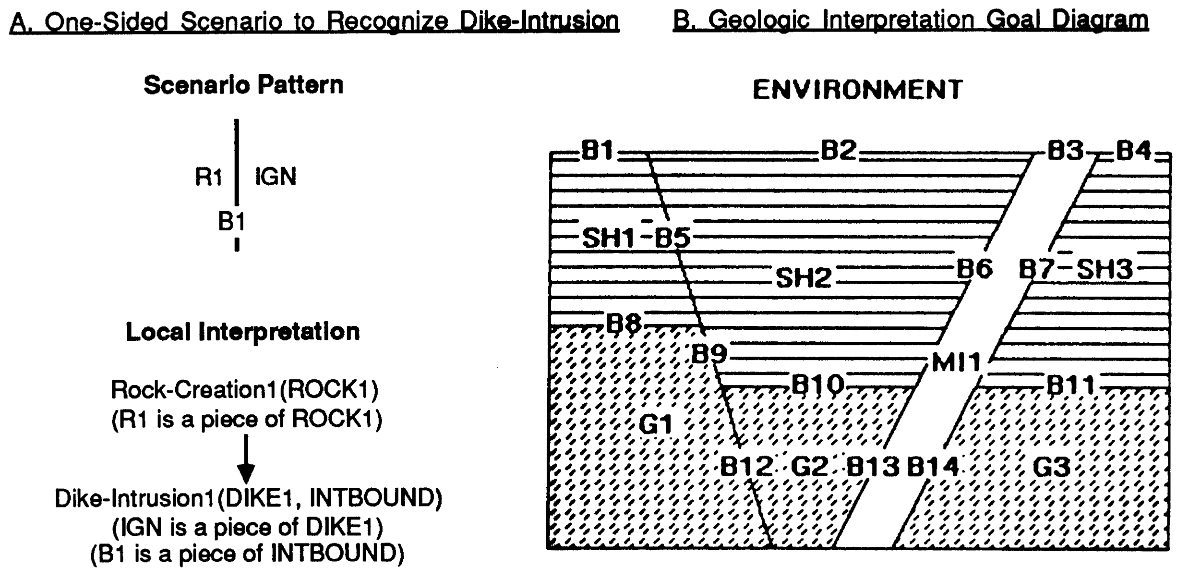

A. One-Sided Scenario to Recognize Dike-Intrusion
Scenario Pattern
R1
IGN
B1
Local Interpretation
Rock-Creation1(ROCK1)
(R1 is a piece of ROCK1)
Dike-Intrusion1(DIKE1, INTBOUND)
(IGN is a piece of DIKE1)
(B1 is a piece of INTBOUND)
B. Geologic Interpretation Goal Diagram
ENVIRONMENT
B1
B2
B3
B4
SH1
B5
SH2
B6
B7
SH3
B8
B9
MI1
B10
B11
G1
B12
G2
B13
B14
G3

Figure 19. Scenario Pattern That Encapsulates Too Few Interactions

Encapsulating too little interaction, on the other hand, yields scenarios that are not sufficiently independent. This can cause the generator to often produce incorrect hypotheses. For example, consider using the scenario in Figure 19a to recognize a dike-intrusion through a formation. In the goal diagram of Figure 19b, the scenario matches the regions **SH2 | MI1** and **MI1 | SH3**, implying that **MI1** intruded through both **SH2** and **SH3**. Using this scenario, however, would cause the generator to hypothesize that two separate rock-creation events occurred because the scenario misses the crucial coupled interaction that **SH2** and **SH3** are pieces of the same original formation. This coupled interaction is encapsulated in our "intrudes-through" scenario, which includes in its pattern both rock-units on either side of the dike and declares in its local interpretation that they were created by the same event.

Both the problem domain and range of problems encountered help to indicate a suitable level for encapsulating interactions. In our domains, Occam's Razor indicates that one should prefer the simplest possible explanations for why interactions arise. For example, a tilted sedimentary rock-unit most commonly arises from the effects of only a single tilt event, so the "tilted-sedimentary" scenario that accounts for such a pattern includes only one tilt event in its local interpretation.

Another indication of how to limit the encapsulations of interactions is to reason about those interactions actually encountered during problem solving. For example, the intrusion pattern **R1 | IGN | R2** is encountered fairly often, but the pattern **R1 | IGN1 | IGN2 | IGN3 | R2** (three intrusions side by side) is rarely seen in geology. This suggests a strategy for creating scenarios based on problem-solving experience: after the debugger solves a problem that the generator handled incorrectly, the system would produce a new scenario that encapsulates the interactions for that problem by analyzing how the debugger repaired the hypothesis. Section 6.1.1 presents an outline for such a learning algorithm.

Our analysis that scenarios encapsulate common patterns of interaction provides insight into how the generator should handle situations in which a goal proposition matches more than one scenario. The generator should prefer the scenario with the most specific pattern that matches, since the smaller patterns encapsulate fewer interactions and, therefore, are more likely to have interference from other scenarios. For example, in cases where both the "intrudes-through" scenario (**R1|IGN|R2**) and the "igneous-under-sedimentary" scenario $\left(\frac{\mathbf{S1}}{\mathbf{IGN}}\right)$ match, the "igneous-under-sedimentary" is less preferable since it is not independent of the event that formed the other side of the intrusion.

While the scenario with the more specific pattern does not always provide the correct interpretation, preferring to use the more specific scenarios is a useful heuristic. The goodness metric used by the search algorithm implicitly embodies this preference for more specific scenarios since it is based on the number of goals achieved and the more specific scenarios tend to achieve more goals.

In cases where two scenario patterns are equally specific, the generator heuristically prefers the more commonly occurring pattern of interaction, which we take to be the one with the fewer events (Occam's Razor). For example, in the problem of Section 2.1 the generator interpreted that granite intruded into shale. Another conceivable interpretation is that the granite was exposed by erosion and then shale was deposited on top of it. Although, given only structural geologic information, it is impossible to tell which interpretation is actually correct, the generator prefers the first interpretation since it involves fewer events and is therefore considered to be more plausible.

3. Test

The tester's role is to verify whether an hypothesis is a valid solution to a problem. The tester used in GORDIUS combines causal and diagrammatic simulation techniques to determine whether the hypothesized set of events can achieve the problem's goal state starting from its initial state. If the hypothesis is found not to be a solution, the tester produces both a list of bugs detected and causal explanations for why the bugs occur.

The tester begins by choosing one of the total orderings consistent with the hypothesis. Testing linear sequences is preferable since it is much more efficient than simulating partial orders. It is also sufficient since our task is to find a single plausible solution, not all (or the best) ones.

The heart of the tester is a domain-independent causal simulator. For each event in the hypothesis, the simulator determines whether the preconditions of the event hold and, if so, updates its model of the world to reflect the changes caused by the event. The simulator can handle both qualitative information (e.g., "the thickness of a rock-unit decreases by some amount") and quantitative information (e.g., "the orientation of a boundary increases by 5°").

We refer to the simulator as "causal" because it embodies a theory of how changes happen in the real world. Basic to the theory is that attributes of objects persist over time, until they are changed by the effects of events. Our models are capable of representing relative, conditional, and quantified effects as well as the creation and destruction of objects. For example, our model of erosion encodes that the thickness decreases for all rock-units whose tops are above the level of erosion, and all rock-units whose bottoms are above that level are destroyed entirely by the effects of erosion.

Since spatial effects such as the splitting of formations due to faulting are difficult to represent and expensive to simulate using the causal simulator, for the geologic domain an additional diagrammatic simulation is performed. The diagrammatic simulator constructs a series of diagrams to represent the spatial effects of geologic events, using a specialized representation for diagrams in which encoding and manipulating topological and geometrical properties of objects is relatively easy and efficient. For example, the effects of erosion are captured by constructing a line in the diagram at the level of the erosion and then erasing the parts of the diagram that lie above the line.

The final step of the tester determines whether the problem's goal propositions hold in the final state of the simulation. For geologic interpretation problems, this involves using a special-purpose diagram matcher to compare the goal and simulation diagrams. The diagram matcher determines both topological and geometrical correspondences between diagrams using a Waltz-like filtering algorithm. It efficiently finds partial matches, especially where one diagram has extra features that do not exist in the other diagram.

To facilitate debugging, the tester constructs causal explanations for why goal propositions do or do not hold. The explanations are in the form of *causal dependency structures*, which are graphs whose nodes represent statements about the world state and whose arcs indicate how their values causally, functionally, or logically depend on the values of other statements. For example, the orientation of a rock-unit at time **t1** might depend on its value at time **t0** plus the fact that the orientation persisted in value between **t0** and **t1**. The causal dependency structures are implemented using a truth-maintenance system to facilitate efficient updating when the hypothesis is modified by the debugger.

The next section continues the example of Section 2.1 by illustrating how GORDIUS tests the generated hypothesis. Section 3.2 describes our causal and diagrammatic simulation techniques and the different specialized representations they utilize. Section 3.3 discusses why simulation is a useful technique for testing interpretations and plans, and analyzes the different strengths and weaknesses of the two types of simulation used — the causal simulator is well suited for constructing causal dependencies, and the diagrammatic simulator is well suited for accurately and efficiently predicting spatial effects.

3.1 Testing a Sequence of Geologic Events

Given the initial hypothesis produced by the generator (Figure 20b), the tester arbitrarily chooses one of its linearizations (Figure 21) to simulate. Testing begins with the causal simulator, which takes each event in the sequence in turn and determines the effects it has on the geologic region — updating its world state to represent the creation and destruction of objects and the alteration of attributes of objects.

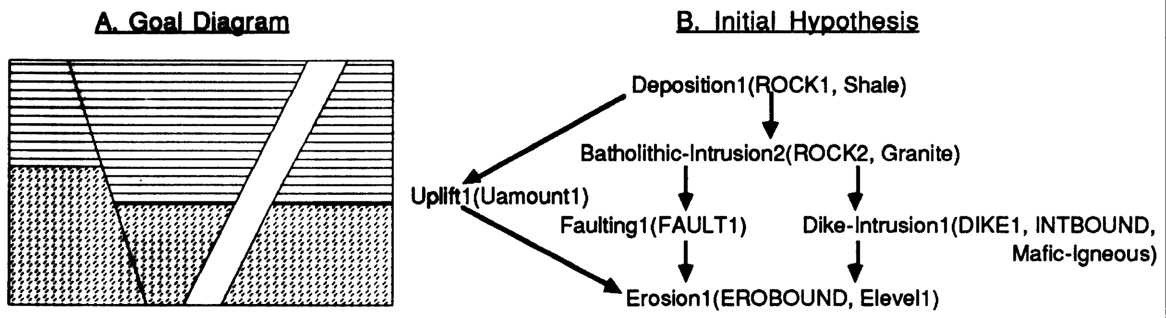

A. Goal Diagram
B. Initial Hypothesis
Deposition1(ROCK1, Shale)
Batholithic-Intrusion2(ROCK2, Granite)
Uplift1(Uamount1)
Faulting1(FAULT1)
Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
Erosion1(EROBOUND, Elevel1)

Figure 20. Initial Hypothesis Produced by the Generator

1. Deposition1(ROCK1, Shale)
2. Batholithic-Intrusion2(ROCK2, Granite)
3. Uplift1(Uamount1)
4. Faulting1(FAULT1)
5. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
6. Erosion1(EROBOUND, Elevel1)

Figure 21. Linearization of Initial Hypothesis, Chosen by the Tester

The initial state given to the simulator consists only of bedrock under sea-level. The simulator determines that the precondition of the first step (**Deposition1**) holds since the surface of the Earth is currently below sea-level. The simulator then updates the world state to reflect the effects described by its declarative model of deposition — that **ROCK1** is created, its composition is shale, its orientation is 0°, it lies on top of the bedrock and along the surface of the Earth, etc.

Simulating **Batholithic-Intrusion2** next, GORDIUS records that a granite **ROCK2** is created, that its intrusional boundary abuts the shale **ROCK1**, that the thickness of the shale decreases due to the intrusion, etc. For **Uplift1**, constraints placed by the generator on **Uamount1**, the uplift parameter, indicate that the event raises the surface of the Earth above sea-level.

Likewise, the rest of the events are simulated, creating a fault boundary, a mafic-igneous formation and intrusional boundary, and an erosional boundary. For these events, parameter constraints made by the generator enable the causal simulator to determine which rock-units are affected. For example, constraints on **Elevel1**, the level of erosion, indicate that both the shale and mafic-igneous formations are affected, with their top points changing and the thickness of the shale decreasing.

Unfortunately, the causal simulation does not determine topological and geometrical effects in enough detail to adequately verify the hypothesis. For example, the models used by the causal simulator are not detailed enough to determine where the dike-intrusion cuts the fault — the hypothesis would be incorrect if the fault were cut within the window of the goal diagram.

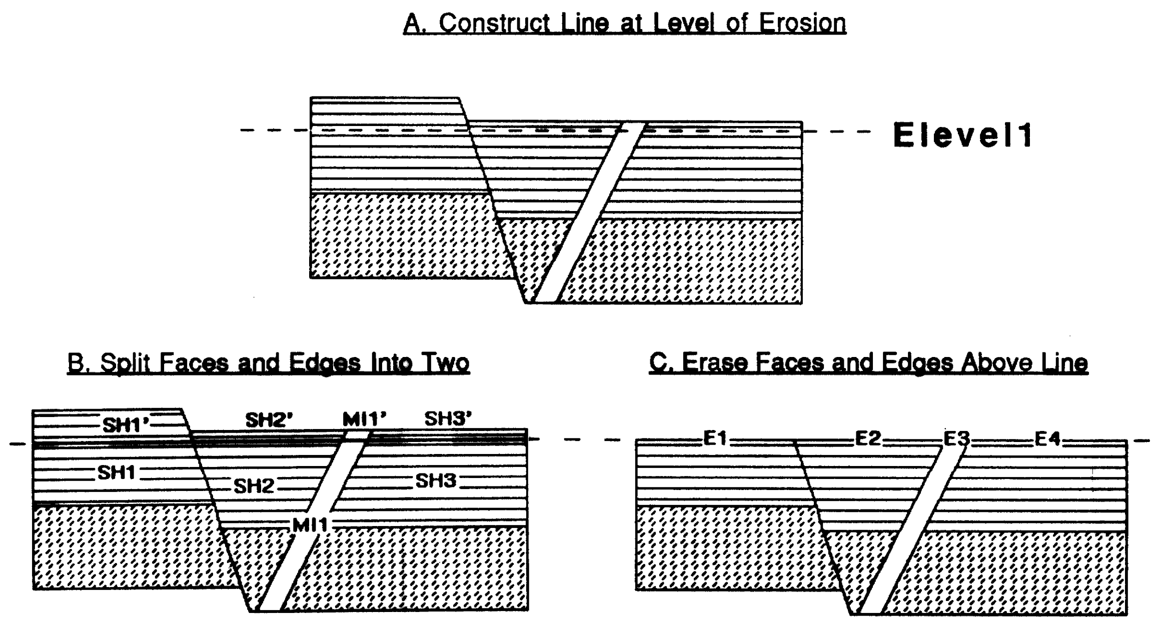

A. Construct Line at Level of Erosion
Elevel1
B. Split Faces and Edges Into Two
SH1'
SH2'
MI1'
SH3'
SH1
SH2
SH3
MI1
C. Erase Faces and Edges Above Line
E1
E2
E3
E4

Figure 22. Diagrammatic Simulation of Erosion

To determine the spatial effects of events more accurately, the tester follows the causal simulation with a diagrammatic simulation. The diagrammatic simulator uses a specialized representation of diagrams that explicitly encodes the topological and geometrical relationships between faces and edges (i.e., rock-units and boundaries), and uses specialized procedures to manipulate the diagram representation. To simulate the effects of erosion, for instance, a line is constructed in the diagram at the height **Elevel1** (Figure 22a) adding new edges and splitting existing faces and edges in two (Figure 22b). The simulator then erases the faces and edges above the line and asserts that the new edges (**E1-E4**) are pieces of the surface of the Earth (Figure 22c).

The numeric parameter values needed for the diagrammatic simulation are obtained by relating the desired parameter values to measurements in the goal diagram. To determine a value for **Elevel1**, for example, GORDIUS measures the height of the top of **SH2** (Figure 20a), since the causal simulation indicates that 1) **Elevel1** equals the height of the top of the shale formation after erosion, and 2) nothing changed that height between erosion and **Plan-end**, the time associated with the goal diagram.

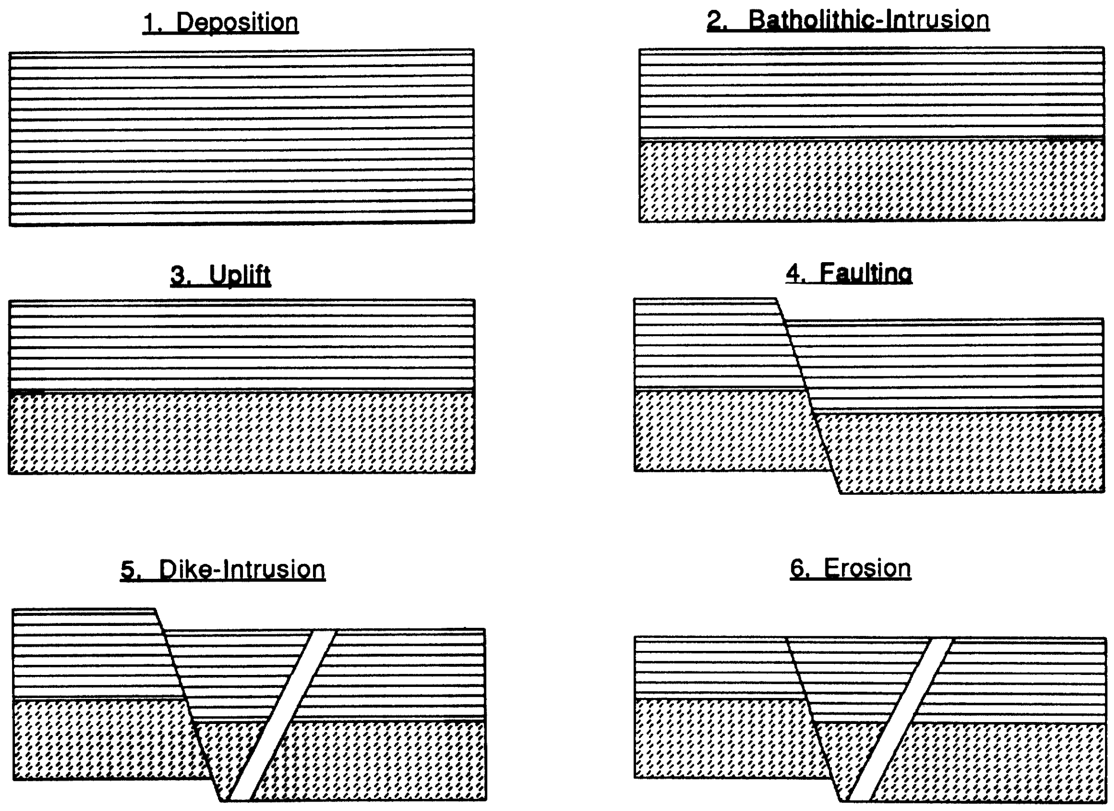


Figure 23. Result of Diagrammatic Simulation of Figure 21

Diagrammatically simulating the initial hypothesis (Figure 21) produces the sequence of diagrams shown in Figure 23. The tester then compares the goal diagram in Figure 20a and the final simulation diagram in Figure 23. A diagram matcher is used to determine correspondences between the faces and edges in the two diagrams, and these correspondences are used to determine whether all the topological, geometrical and compositional goals of the problem are met. In this case, the simulation diagram is judged to achieve all the goals, so the tester reports that at least one linearization of the initial hypothesis (Figure 21) is a plausible interpretation of the problem (Figure 20a).

If this were always the outcome, the story could end here. Occasionally, however, the generator produces an invalid hypothesis. Such hypotheses must be detected by the tester and reported to the debugger, together with an explanation for why the test failed.

To illustrate this, suppose we alter the "erosion-ends-boundary" scenario by removing all mention of the occurrence of uplift. In this case, the generator produces the initial hypothesis in Figure 24a and the tester attempts to simulate the linearized sequence of Figure 24b. The causal simulation proceeds until it reaches the the erosion event. At this point, the tester detects a bug — **Erosion1** cannot occur because our models indicate that erosion occurs only above sea-level, while the simulation predicts that the surface of the Earth is currently below sea-level (since **Deposition1** occurred under water and no subsequent event acted to raise the surface).

<u>A. Initial Generated Hypothesis</u>

$$\begin{tikzcd}
& \text{Deposition1(ROCK1, Shale)} \arrow[d] & \\
& \text{Batholithic-Intrusion2(ROCK2, Granite)} \arrow[dl] \arrow[dr] & \\
\text{Faulting1(FAULT1)} \arrow[dr] & & \text{Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)} \arrow[dl] \\
& \text{Erosion1(EROBOUND, Elevel1)} &
\end{tikzcd}$$

<u>B. Linearized Sequence For Testing</u>

1. Deposition1(ROCK1, Shale)
2. Batholithic-Intrusion2(ROCK2, Granite)
3. Faulting1(FAULT1)
4. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
5. Erosion1(EROBOUND, Elevel1)

Figure 24. Hypothesis Generated by Incomplete Scenarios

In addition to detecting the bug that erosion cannot occur, the tester produces a causal dependency structure that explains why the bug arises. The (partial) causal dependency structure in Figure 25 indicates that the bug depends on the belief that the Earth's surface is below sea-level at the start of erosion. This, in turn, depends on knowledge about the value of sea-level and its relation to the value of the height of the top of the Earth's surface at the start of the erosion. Similarly, the height of the surface depends on various factors, including the belief that the height of the top of the surface persists in value from the end of shale deposition to the start of erosion.

Chapter 4 discusses how the debugger actually uses these dependency structures to repair hypotheses. Here, we examine how the tester works and how causal dependency structures are constructed.

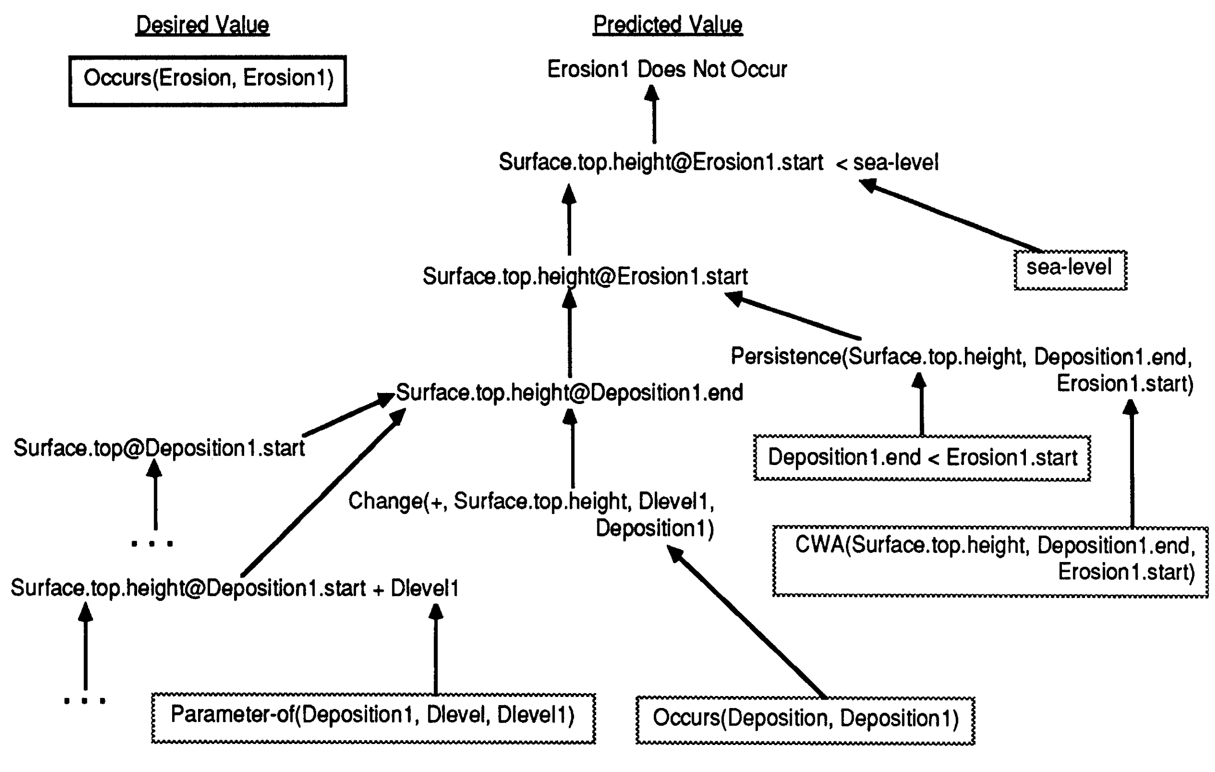

Desired Value
Predicted Value
Occurs(Erosion, Erosion1)
Erosion1 Does Not Occur
Surface.top.height@Erosion1.start < sea-level
sea-level
Surface.top.height@Erosion1.start
Persistence(Surface.top.height, Deposition1.end, Erosion1.start)
Surface.top.height@Deposition1.end
Surface.top@Deposition1.start
Deposition1.end < Erosion1.start
Change(+, Surface.top.height, Dlevel1, Deposition1)
CWA(Surface.top.height, Deposition1.end, Erosion1.start)
Surface.top.height@Deposition1.start + Dlevel1
Parameter-of(Deposition1, Dlevel, Dlevel1)
Occurs(Deposition, Deposition1)

Figure 25. Partial Causal Dependency Structure for the Bug that Erosion Cannot Occur

3.2 The Testing Algorithm

The next four sections describe the representations and reasoning techniques used in the four stages of the GORDIUS tester: 1) a totally ordered sequence is chosen to be tested; 2) a causal simulation is performed to predict the effects of each event; 3) for the geologic domain, a diagrammatic simulation is performed to gain more accurate information about the spatial effects of the events; 4) the simulation state is compared against the goal state.

3.2.1 Linearizing Hypotheses

The first stage of the tester chooses one total ordering consistent with the partially ordered events of the hypothesis. For each unordered pair of events in the hypothesis, the tester adds an assumption that one (chosen arbitrarily) actually precedes the other.

This linearization is done for efficiency. For the types of representations needed to encode our geologic domain, the computational complexity of simulation is exponential in the number of events for partial orders, but is only polynomial for total orders (see [Chapman]).

Testing linear sequences is also a very reasonable strategy for GORDIUS. This can be seen by analyzing three categories of partial orders — ones in which all, none, or some of the linearizations are correct. If all are correct then any linearization will obviously do, since the task is only to find a single plausible solution. Our experimental evidence shows that since the generator usually produces correct hypotheses, this is the most commonly occurring case. If all linearizations are incorrect, it does not matter which sequence is chosen since it will have to be debugged anyway.

The problematic case is where only some linearizations are solutions, since debugging will be needed if an incorrect linearization is chosen to be tested. Even in this case, however, testing and debugging a linear sequence is usually more efficient than testing a partial order, since typically only a fraction of the exponential number of possible linearizations need to be tested and debugged before a solution is found. This is often the case because the dependency structures produced by the tester enable the debugger to focus on replacing only those orderings that actually contribute to the bugs.

Figure 26 illustrates a case where the choice of linearization matters. The example is similar to the one in Section 3.1, except that the window of the goal diagram is extended downwards. The generator produces the same partially ordered hypothesis with the uplift, dike-intrusion, and faulting left unordered (see Figure 20b). In this example, having the faulting precede the dike-intrusion is a solution (see Figure 26b) but the alternative ordering is not — the diagrammatic simulation shows that an extra piece of mafic-igneous appears within the window of the goal diagram (see Figure 26c).[1]

As described more fully in Section 5.1.3, the debugger analyzes the tester's explanation for why the extra piece appears and determines that the dike-intrusion and faulting events should be reversed. Note that the ordering of the uplift event is unaffected since the bug is independent of that particular ordering.

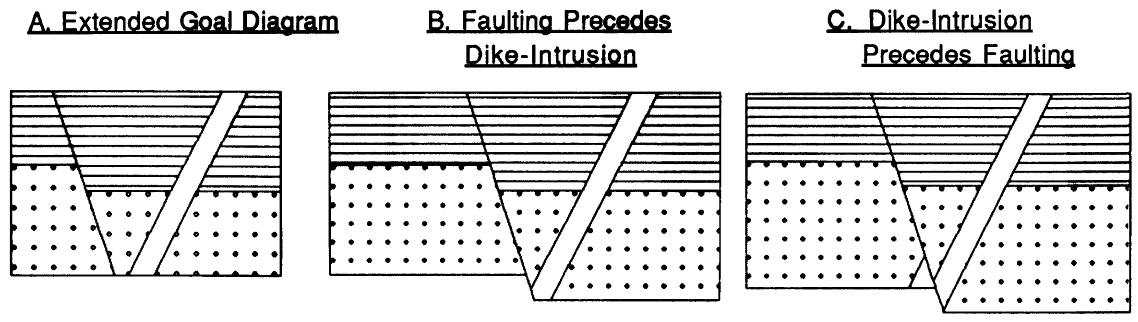

A. Extended Goal Diagram
B. Faulting Precedes Dike-Intrusion
C. Dike-Intrusion Precedes Faulting

Figure 26. Consequences of Two Choices for Linearizing the Hypothesis

3.2.2 Causal Simulation

The causal simulation algorithm itself is quite simple. The simulator considers each event in the sequence in turn and checks whether its preconditions hold in the current state. If some preconditions do not hold, the simulator reports the bug that the event cannot occur, together with causal dependency structures describing why the preconditions do not hold.

If all preconditions hold, the simulator updates the world state to reflect the effects of the event. It also maintains dependencies to indicate how the current state enables the (conditional) effects to happen and how the effects in turn change the world state.

The simulator uses declarative event models to determine whether events can occur and what effects they have. Event models are represented by four fields — parameters, preconditions, effects, and constraints.

[1] In the example of Section 3.1, the ordering of the dike-intrusion and faulting is truly indeterminate given the limited window of the goal diagram.

```
Parameters:  Boundary : Boundary (Created)
             Elevel : Real
Preconditions: Surface.top.height@Erosion.start > sea-level
Effects:
   Change(=, Boundary.orientation, 0, Erosion)
   Change(=, Boundary.side-1, {Environment}, Erosion)
   Change(=, Boundary.side-2, Ero-Surface(Elevel, Surface, Erosion.start), Erosion)
   (For-all ($ru : rock-unit)
       (If [ Exists-at($ru, Erosion.start) and
             $ru.top.height@Erosion.start ≥ Elevel  and  $ru.bottom.height@Erosion.start < Elevel]
           [ Change(-, $ru.thickness, Erosion-of(Elevel, $ru, Erosion.start), Erosion) and
             Change(=, $ru.top, EroFn1(Elevel, $ru, Erosion.start), Erosion)]))
   (For-all ($bd : boundary)
       (If [ Exists-at($bd, Erosion.start) and
             $bd.top.height@Erosion.start ≥ Elevel  and  $bd.bottom.height@Erosion.start < Elevel]
           Change(=, $bd.top, EroFn1(Elevel, $ru, Erosion.start), Erosion)))
   (For-all ($gf : geologic-feature)
       (If [ Exists-at($gf, Erosion.start) and $gf ≠ Surface and
             Elevel ≤ $gf.bottom.height@Erosion.start]
           Destroyed($gf, Erosion.end)))
   Change(=, Surface.top, EroFn2(Elevel, Surface.top@Erosion.start), Erosion)
   Change(=, Surface.bottom, EroFn2(Elevel, Surface.bottom@Erosion.start), Erosion)
   Change(=, Surface.orientation, EroFn3(Elevel, Surface, Erosion.start), Erosion)
   Change(=, Surface.side-2, Ero-Surface(Elevel, Surface, Erosion.start), Erosion)
Constraints :
   Elevel > sea-level
   Surface.bottom.height@Erosion.end ≤ Surface.bottom.height@Erosion.start
   Surface.top.height@Erosion.end = Elevel
   Surface.bottom.height@Erosion.end ≤ Elevel
   (If Surface.bottom.height@Erosion.start ≥ Elevel
       Surface.bottom.height@Erosion.end = Elevel)
   (For-all ($ru : rock-unit)
       (If  [ Exists-at($ru, Erosion.start)  and
              $ru.top.height@Erosion.start ≥ Elevel  and  $ru.bottom.height@Erosion.start < Elevel]
            [ $ru ∈ Ero-Surface(Elevel, Surface, Erosion.start.start) and
              $ru.top.height@Erosion.start.end = Elevel]))
```

EroFn1, **EroFn2**, **Erosion-of**, and **Ero-Surface** are user-defined functions representing different aspects of the erosion event (e.g., **Erosion-of(Elevel, ru, t)** represents the amount of **ru** eroded away; it is constrained to be non-negative and less than the thickness of **ru**).

Figure 27. Event Model of Erosion

The parameters field describes the formal names and types of the event's parameters. For example, the parameters of erosion (Figure 27) include **Elevel**, a real-valued quantity representing the level to which the erosion occurs, and **Boundary**, the erosional boundary created by the event. The declaration that a parameter is created by an event provides important information to the system. For instance, the concept of necessarily unifiable events, used by the generator to compose local interpretations, depends on knowing which objects an event creates.

The preconditions field contains a set of propositions that must hold in order for the event to occur. For example, the precondition for erosion is that the height of the top of the Earth's surface at the start of the erosion must be above sea-level (Figure 27); the preconditions of the blocks-world **Puton** event are that both the source and destination blocks must be clear (Appendix C).

The effects field describes the changes an event can have and the conditions under which they can happen. The events are all *discrete* in that their effects can be used to predict the state of the world after an event occurs but say nothing about what happens during the occurrence. We chose to use discrete event models because they are sufficient for the domains explored.

In order to encode complex domains, such as geology and semiconductor fabrication, we extended traditional discrete action models, which represent effects as simple propositions (e.g., [Fikes], [Sacerdoti], [Wilkins]). The extensions are significant in that they allow:

1. Effects that are expressed in relative terms (e.g., uplift increases the height of objects by the amount of uplift).
2. Effects that are conditionalized (e.g., if a rock-unit is on the surface then its thickness decreases as a result of erosion).
3. Effects that are universally quantified (e.g., tilt changes the orientation of all boundaries).
4. Creation and destruction of objects (e.g., deposition creates a sedimentary formation; erosion destroys rock-units whose bottom-most point is above the level of erosion).

Effects are described using statements of the form:

Change(type, object.attribute, magnitude, event).

The **change** statement is interpreted to mean that the **attribute** of the **object** is affected by the **event**. **Type** and **magnitude** together determine the value of the attribute at the end of the event. If **type** is = then the attribute's value is equal to **magnitude**. For example, the statement **Change(=, Boundary.orientation, 0, Erosion)** represents the effect that the orientation of the erosional boundary is horizontal. Relative changes are also represented using this syntax. If **type** is + or -, the value of the attribute at the end of the event is determined relative to its starting value. For example, **Change(-, $ru.thickness, Erosion-Of(...), Erosion)** encodes the effect that erosion decreases the thickness of rock-units by some amount. If the type of **attribute** is a set, such as the set of rock-units along either side of a boundary, then + and - are interpreted as set insertion and deletion, respectively.

Two special types of change statements are used to indicate that objects are created and destroyed. **Destroyed(object, time)** indicates that after **time** the **object** ceases to exist. **Created((object, type, time) consequent)** is a type of existential statement indicating that a new **object** of **type** is created at **time**, and that the **consequent** statement holds after the object is created. All parameters declared to be created by an event are treated this way.

Conditional effects are represented using statements of the form:

(If antecedent consequent),

which indicates that the **consequent** holds if the **antecedent** holds. **Antecedent** can be either a single proposition, a conjunction, or a disjunction of propositions. **Consequent** is limited to being one or a conjunction of the following types of statements: **Change**, **Created**, **Destroyed**, **If**, and **For-all**. Quantification of effects is described using the statement:

(For-all (var : type) consequent),

which indicates that the **consequent** holds for all objects of **type**, where **consequent** has the same form as in conditional statements.

The fourth field of an event model consists of constraints on the values of parameters and the magnitudes of effects. For example, the erosion model constrains the level of erosion (**Elevel**) to be greater than sea-level, and it constrains the height of the bottom of the surface at the end of erosion to be no greater than its height before the erosion occurred.

As mentioned above, the causal simulator uses the event models to determine whether events can occur and what effects they have. It accomplishes this by substituting the event's parameter bindings for the formal parameters in the models and evaluating the resulting preconditions, effects, and constraint statements.

Much of the power and flexibility of the causal simulator stems from its technique for evaluating statements. Associated with each type of statement is an evaluation method that can determine the value of a statement given the current state of the simulation. The methods also know how to update the various specialized representations used by the system (described below) when a statement is asserted to have a particular value.

All the evaluation methods are local in that a statement's value can be determined just from the values of its arguments. The methods also all record dependencies that indicate how the value was derived. The dependencies recorded are causal in that the evaluation methods reflect our model of how the physical world works. The dependencies facilitate incremental retraction and update of inferences using a TMS mechanism and are used extensively by the debugger and the diagrammatic simulator. The next few pages describe the evaluation methods and specialized representations implemented in GORDIUS.[2]

Arithmetic Relations and Operations — Numeric quantities are represented and reasoned about using the Quantity Lattice [Simmons, 86], which integrates information about ordinal relationships between quantities ($<,\leq,>,\geq,=,\neq$) together with information about the real-valued range within which the value of a quantity lies (e.g., [0...2]). The Quantity Lattice employs multiple inference methods to determine relationships between quantities, including using graph-search based on the transitivity of relations, and reasoning about whether the ranges of quantities overlap. For example, from the assertions **A < B** and **B ≤ C** the Quantity Lattice can infer that **A < C** by transitivity. From **D < 1.36** and **E > 2.56** the system can infer that **D < E** since the upper bound of **D** is less than the lower bound of **E**.

The Quantity Lattice also handles expressions involving arithmetic operations (e.g., **+**, **-**, *****, /, **abs**, **sin**), computing the value of an expression using interval arithmetic on its arguments. Evaluating arithmetic expressions may also constrain ordinal relationships between quantities. For example, in evaluating the expression **A + 2** the Quantity Lattice asserts that the value of the sum is greater than **A** since it knows that adding a positive amount to any quantity increases its value.

[2] [Simmons, 83] describes some of the specialized representations in more detail.

Time — Since change is measured relative to time, the explicit representation of time is crucial in reasoning about how the world changes. In our system, time points are primitive and are represented by quantities in the Quantity Lattice. An interval of time **I** is represented by its start and end points, **I.start** and **I.end**, respectively. The linearity of time is encoded by assuming that **I.start < I.end** for all intervals.

Objects, Attributes, and Temporal References — Objects are represented as collections of attributes, where the attributes represent not single values but *histories* of values over time. A history is represented as a sequence of time intervals, alternating between *dynamic* and *quiescent* intervals. Dynamic intervals represent times during which an event is changing the value of the attribute; quiescent intervals represent times when the attribute persists in value.

Temporal objects, such as rock-units and boundaries, also have a temporal extent (their **start** and **end** times). The statement **Exists-at(object, time)** is true if **time** falls between the **start** and **end** of the **object**. The truth of this statement depends on the facts that **object** is created before **time** and is not destroyed before **time**. In the absence of conflicting information, GORDIUS assumes that objects continue to persist in existence indefinitely after they are created.

Temporal references (of the form **object.attribute@time**) represent the value of the **object**'s **attribute** at **time**. For example, **R1.orientation@t0** refers to the orientation angle of rock-unit **R1** at time **t0**. A temporal reference is evaluated by searching the history time-line of the **attribute** to find the interval in which the **time** point falls. If **time** falls within a dynamic interval (i.e., **I.start < time < I.end**) the value of the attribute is **unknown**, since the use of discrete event models prevents GORDIUS from predicting what happens while an event is occurring.

If **time** falls at the end of a dynamic interval (or at the start of the next quiescent interval), the value of the temporal reference depends on the **change** statement that produced the dynamic interval (see below). If **time** falls within a quiescent interval (or at the start of a dynamic interval), the value depends on both the value at the start of the interval and on a persistence statement that the attribute does not change from the start of the quiescent interval until **time**.[3] The persistence statement, in turn, is supported by the statement that the start of the quiescent interval precedes **time** and by the closed-world assumption that no known event changes the attribute during that interval.

The value of a temporal reference also depends on the value of the **object** when it itself is a temporal reference. For example, the expression **(Surface.top@t1).height@t2** (i.e., the height at time **t2** of the point that was the top of the surface at time **t1**) partly depends on the value of **Surface.top@t1**. This dependency information enables the system to infer that it can change the value of **(Surface.top@t1).height@t2** either by changing the height (e.g., by uplift) or by changing the surface's top-most point (e.g., by erosion).

Change Statements — When a statement of the form **Change(type, object.attribute, magnitude, event)** is asserted, GORDIUS updates the history time-line of the **attribute** by adding a dynamic interval whose extent equals that of the **event**. The value of the **attribute** at the end of the dynamic interval is constrained to be equal to the value determined by the **type** and **magnitude**, as described previously. The truth of the special change statement **Destroyed(object, time)** is used to support the truth of the equality **object.end=time**. Similarly, the **Created** change statement supports the equality **object.start=time**. Together, these equalities are used to support the **Exists-at** statement described above.

Logical Propositions — GORDIUS supports propositional logic, including the standard boolean connectives **and**, **or**, **not**, $\Rightarrow$ (implication), and **iff** (equivalence), using the RUP truth-maintenance system [McAllester]. Inferences based on the truth values of propositions are derived using natural deduction in a forward-chaining manner. For example, asserting the propositions **On(A, B, t1)** and **On(A, B, t1) $\Rightarrow$ not Clear(A, t1)** would cause the system to infer that **Clear(A, t1)** is false.

Quantification — GORDIUS also supports limited forms of quantification. The statement **(For-all (var : type) consequent)** is evaluated by finding all objects of **type** and, for each object, creating a new statement by substituting the name of the object for

[3] This does not preclude the fact that the **attribute** might still persist past **time**.

var in the **consequent** statement. The method then asserts that the universal statement is equivalent to the conjunction of the newly created statements plus the closed-world assumption that no other object of **type** is known to exist. If the system subsequently discovers a new object of **type**, the closed-world assumption is retracted and a new conjunction of statements is created. The existential statement **(Exists (var : type) consequent)** is handled similarly.

Set Relations and Operations — The system currently has a simple method for reasoning about set equality and set membership.[4] The three types of assertions about sets currently handled are: 1) two sets are equal, in which case they are constrained to have the same members; 2) an object is (or is not) a member of a set (e.g., $\mathbf{x} \in \mathbf{S}$); 3) a set is closed, that is, its only members are those already known to the system. Set closure is useful for evaluating membership and set equality relations. For example, if **RockSet** is closed and there are no explicit assertions that **Rock1** is a member of **RockSet**, then GORDIUS can infer that $\mathbf{Rock1} \notin \mathbf{RockSet}$.

New sets can be constructed using the operations **Union**, **Intersect**, **Difference**, **Set-Insert** and **Set-Delete** (the last two operations add and delete single elements from a set). The system maintains the appropriate membership constraints among sets as their members become known. For example, asserting that **E1** is a member of **Intersection(Set1, Set2)** causes the system to assert that **E1** is also a member of both **Set1** and **Set2**.

Definitions — GORDIUS provides the capability for the user to define new relations and functions. A relation is defined by giving the names and types of its arguments and (optionally) a logical expression that defines the relation. In the blocks-world, for example, the relation **Clear(b, t)** is defined as: **b.top@t = {}** (i.e., block **b** is clear at time **t** if the set of blocks on top of **b** is empty).

To evaluate an instance of a defined relation, the system asserts that it is equivalent to its definition and then uses the methods described above to evaluate the definition. For relations that do not have an associated definition, the user can specify a default truth

[4] We have recently implemented a more sophisticated set reasoning system [Wellman & Simmons], but currently it is not integrated into GORDIUS.

value to use. In such cases, the truth value of the relation is presumed to depend on the values of each of its arguments.

Similarly, a function is defined by specifying its arguments, the type of function's range, and (optionally) an expression for computing the function's value. To evaluate a defined function, the system asserts that its value equals its definition. For functions without a definition, such as the **Erosion-of** function in Figure 27, the system default action is to create a new object of the range type and assert that the function's value is equal to that new object.

To illustrate the evaluation of statements, consider the following quantified and conditional effect (taken from Figure 27) which has been instantiated for an erosion event **Erosion1** that erodes to depth **Elevel1**:

```
(For-all ($ru : rock-unit)
  (If [ Exists-at($ru, Erosion1.start) and
       $ru.top.height@Erosion1.start ≥ Elevel1 and
       $ru.bottom.height@Erosion1.start < Elevel1]
    Change(-, $ru.thickness, Erosion-of(Elevel1, $ru, Erosion1.start), Erosion1))).
```

The effect states that for each rock-unit existing at the start of the erosion, where **Elevel1** falls between the top-most and bottom-most points of the rock-unit, its thickness decreases by some amount (i.e., the rock-unit is partially eroded). **Erosion-of**, representing the amount decreased, is a user defined function whose value is constrained to be non-negative and less than the thickness of **$ru**.

To evaluate the above statement, GORDIUS finds all objects of type **Rock-Unit** and creates a conjunction of **If** statements by substituting the name of each rock-unit for the variable **$ru**. GORDIUS asserts that the **For-all** statement is logically equivalent to this conjunction plus the closed-world assumption that all objects of type **Rock-Unit** are known. When the simulator asserts that the **For-all** statement is true, the deductive methods in RUP infer that the conjunction is true and therefore that each individual **If** statement is true.

We focus now on one particular conjunct — that of rock-unit **SH1**:

```
(If [ Exists-at(SH1, Erosion1.start) and
     SH1.top.height@Erosion1.start ≥ Elevel1 and
     SH1.bottom.height@Erosion1.start < Elevel1]
  Change(-, SH1.thickness, Erosion-of(Elevel1, SH1, Erosion1.start), Erosion1))).
```

The system evaluates each expression in the antecedent of the **If** statement. The truth of the **Exists-at** statement is determined by checking whether **Erosion1.start** falls within the temporal extent of **SH1**. The other two antecedent expressions are evaluated using the Quantity Lattice to determine the relationships between the parameter **Elevel1** and the temporal references:

SH1.top.height@Erosion1.start and **SH1.bottom.height@Erosion1.start.**

If all expressions in the antecedent evaluate to true, GORDIUS deduces that the **Change** statement is also true, causing it to update the **thickness** history time-line of the **SH1** rock-unit by adding a dynamic interval that has the same extent as **Erosion1**. Finally, GORDIUS constrains the eroded value of the thickness of **SH1** by asserting:

```
SH1.thickness@Erosion1.end =
        SH1.thickness@Erosion1.start - Erosion-Of(Elevel1, SH1, Erosion1.start).
```

Since the value of the **Erosion-Of** function is defined to be non-negative, the Quantity Lattice infers that the thickness of **SH1** after the erosion is no greater than its thickness before the event occurred.

3.2.3 Diagrammatic Simulation

For geologic interpretation, a diagrammatic simulation is performed to produce the additional spatial information needed to adequately test hypotheses. The simulator uses a specialized diagram representation and manipulation procedures to efficiently determine detailed spatial effects of the geologic events.[5]

Our representation of diagrams is based on the wing-edge representation of [Baumgart]. Originally designed for three-dimensional modeling in computer vision, we have adapted it for two-dimensional diagrams. The primitive objects in the wing-edge representation are *edges, faces* and *vertices*. Edges and faces encode the topology of the diagram: attributes of an edge include 1) the two faces on either side of it, 2) its two end vertices, and 3) its connecting edges; attributes of a face include the edges on its perimeter. The geometry of a diagram is represented by the vertices, which encode their (X,Y) coordinate positions.

The wing-edge representation is well suited to representing and reasoning about spatial changes to geologic objects for several reasons. First, the primitive objects used in the representation —

[5] More detail is found in [Simmons, 83].

faces, edges, and vertices — have a natural correspondence with the primitive objects used in the geologic representation — rock-units, boundaries, and geologic points. Second, the representation facilitates computation of the spatial relationships (e.g., "above") and metric properties (e.g., "orientation") that are needed for geologic interpretation.

For the purpose of testing geologic hypotheses, the most important advantage of the wing-edge representation is the ease with which spatial changes can be simulated. The diagrammatic simulator uses a set of low-level operations that modify the topology of wing-edge structures. These operations can add and erase edges, split faces or edges into two parts, and merge two faces or edges into one. The operations are extremely efficient because they involve only local changes to the wing-edge structure. For example, splitting a face involves only changes to the face object itself and to the edges on its perimeter. This is exactly what one wants from a spatial representation — local spatial changes involve only local representational changes.

The initial state given to the diagrammatic simulator is a diagram consisting of a single line, representing the surface of the bedrock. For each hypothesized event, the simulator applies a procedure that takes as input a diagram and numeric parameter values and modifies the diagram to reflect the spatial changes of the event.

The simulation procedures are quite simple for uplift, subsidence, and tilting, since those events involve only geometrical changes. The effects of the events can be simulated merely by adding some value to the **(X,Y)** coordinates of the diagram's vertices. For instance, the uplift procedure adds a constant to each **Y** coordinate.

The rest of the geologic events (erosion, deposition, dike-intrusion, batholithic-intrusion, and faulting) all have topological as well as geometrical effects. The procedures for simulating these events all rely on a method for constructing lines in the diagram. For example, erosion is simulated by constructing a line at the depth of erosion and erasing all parts of the diagram above the line; dike-intrusion is simulated by constructing two parallel lines and erasing all non-boundary edges that fall between the lines.

To give a better feel for the types of manipulations used by the diagrammatic simulator, we present the following iterative algorithm used for constructing a line:

1. Find the boundary edge along the top or left-hand side of the diagram that intersects the line being constructed (edge **E1** at point **P1** in Figure 28). Pick the interior face abutting that edge (face **F1** in Figure 28).
2. Starting with edge **E1**, examine in order the edges along the perimeter of the face to find the edge that intersects the line at a point[6] that is closest to, and below or to the right of, point **P1** (edge **E2** at point **P2**). Construct an edge **E'** from point **P1** to **P2**, splitting face **F1** into two pieces, **F1** and **F1'**.
3. Starting with edge **E2** and examining the perimeter of **F2**, the face on the other side of **E2** from face **F1**, repeat steps 2 and 3 until no more intersection points are found.

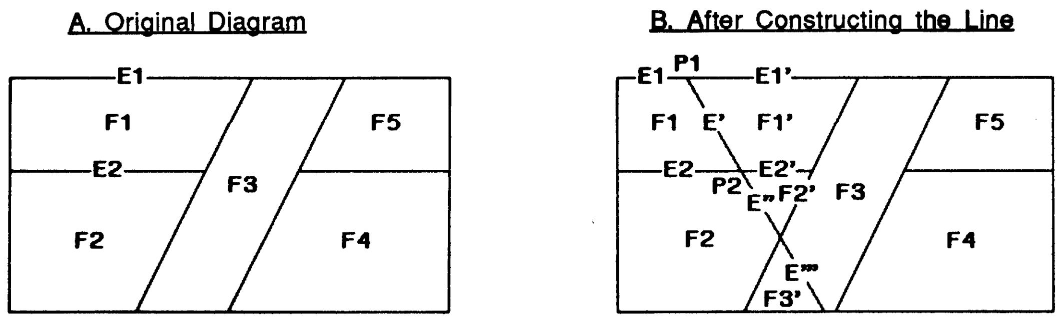


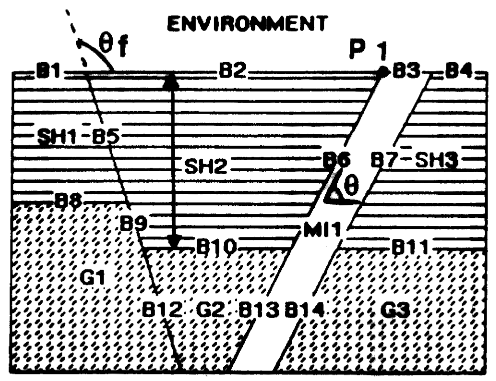
Figure 28. **Constructing** a Line in a Diagram

The line construction method also maintains associations between topological changes in the diagram and geologic changes in the object/attribute/history representation recorded by the causal simulator. For example, when a face is split in two, adding one new face, the line construction method associates the new face with a newly created rock-unit and records that the new rock-unit is a piece of, and has the same composition as, the rock-unit associated with the face that was split. For example, if face **F1** in Figure 28a is associated with shale rock-unit **SH1**, the line construction method creates a new rock-unit (say **RU1**), associates **F1'** with **RU1**, and asserts that **RU1** is a piece of **SH1** and that it is composed of shale.

[6] There may be more than one intersection point if the face is concave.

One difficulty is that the constraints on parameter values produced by the generator are typically only qualitative and therefore not precise enough for an accurate diagrammatic simulation. Our solution is for GORDIUS to obtain numeric parameter values by making measurements in the goal diagram and relating the measurements to the desired parameter values by tracing through the dependencies recorded during causal simulation.

For example, the procedure that simulates dike-intrusion needs to know the location of the intrusion in order to place it correctly in the diagram. The location is given by the equation for the intrusion's centerline, which is parameterized by the slope **M** and the y-intercept **B** of the line. GORDIUS determines that **M** is equal to the orientation of **MI1** in the goal diagram (θ in Figure 29) since the causal simulator records that at the time of the intrusion the orientation of **MI1** was **M** and that the orientation was not affected by subsequent events.



1. Deposition1(ROCK1, Shale)
2. Batholithic-Intrusion2(ROCK2, Granite)
3. Uplift1(Uamount1)
4. Faulting1(FAULT1)
5. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
6. Erosion1(EROBOUND, Elevel1)

Figure 29. Measuring From the Diagram

Similarly, the y-intercept **B** can be determined by picking a point **P1** on the centerline and solving **B=Y0-M*X0**, where **(X0,Y0)** are the coordinates of **P1** at the time of the intrusion. GORDIUS reasons that, due to the faulting event, points on **MI1** have moved downwards and to the right since the dike-intrusion occurred. Thus, **(X0,Y0)**, the coordinates of **P1** at the time of intrusion, are related to **(Xm, Ym)**, the current coordinates as measured in the goal diagram, by the equations:

$$X0 = Xm - DS*\cos(\theta_f) \text{ and } Y0 = Ym - DS*\sin(\theta_f).$$

θ_f is the (measured) orientation of the fault line and **DS** is the fault slippage, which can be determined, for instance, by measuring the length of edge **B9** in Figure 29.

For some parameters, the system can determine only ranges for their values. For example, GORDIUS can determine that the amount of shale deposition is greater than the thickness of **SH2** (Figure 29) since the causal simulation indicates that erosion decreased the original thickness of the shale formation. How much greater the original thickness was, however, cannot be determined. In such cases, the system chooses an arbitrary value within the known range of the parameter and propagates constraints so that other parameter values are consistent with the chosen value. For example, in choosing to make the shale deposition 12 meters greater than the measured amount of **SH2**, GORDIUS constrains the value of **Elevel1** (the level of erosion) such that in the final simulation diagram the thickness of shale remaining after erosion will match its thickness as measured in the goal diagram.

3.2.4 Matching Diagrams

The final stage of the tester determines whether the simulation achieves all the goals of the problem. The system evaluates each goal state proposition in the context of the simulated world state. If none of the goals evaluate to false, the hypothesis is presumed to be a plausible solution to the problem; otherwise the debugger is invoked to repair the hypothesis. Currently, the system accepts hypotheses as solutions even if the truth of some goal propositions is unknown. A more complete algorithm, however, should try to gather more information to determine whether the goals are in fact achieved.

This simple means of matching the goal and simulated states is used for all the domains we explored. Before evaluating the goals for the geologic domain, however, the system must determine correspondences between the parts of the goal diagram and simulation diagram, since the diagrammatic simulator does not maintain such correspondences itself.

We have implemented a diagram matcher that determines topological and geometrical correspondences between the faces and edges in diagrams. Importantly, for our purposes, the algorithm handles diagrams that only partially match, usually producing matches that agree with commonsense. In Figure 30, for instance, the matcher determines that **SS1** corresponds to **R1**, **SS2** to **R3**, **MI1** to **R2**, **MI2** to **R4**, **SS3** to **R5**, and similarly for the edges in both diagrams. In addition, the matcher notes that faces **R6** and **R7** in the simulation diagram do not correspond to any object in the goal diagram.[7]

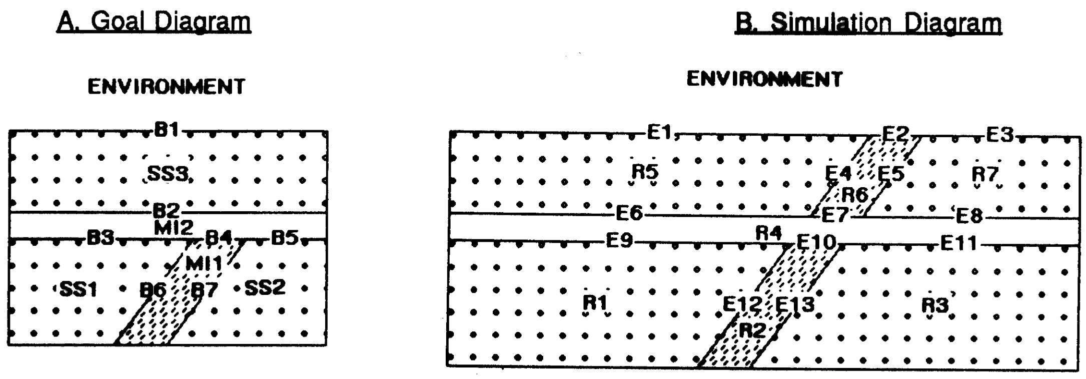


Figure 30. **Partially Matching Diagrams**

Our diagram matcher uses a **Waltz**-like filtering algorithm that enforces the constraint that objects in two diagrams correspond if the objects they abut also correspond. For example, in Figure 30 edge **B6** is held to correspond to **E12** if faces **SS1** and **MI1** correspond to **R1** and **R2**, respectively.

The matching algorithm starts by labeling each face (and edge) in the goal diagram with the set of all faces (or edges) in the simulation diagram, and *vice versa*. Three pieces of geologic knowledge are then applied to constrain the labels. First, the faces representing the **Environment** in both diagrams are constrained to correspond. Second, since corresponding rock-units must have the same composition, the algorithm removes from the label of each face those faces that represent rock-units with different composition. For example, the label for **SS3** is constrained to be **{R1,R3,R5,R7}**. Third, the edges forming the windows of the diagrams (the side and bottom edges) are removed from all labels since they do not correspond to real geologic entities.

The matching algorithm proceeds to enforce the topological constraint that if two objects abut in one diagram they must

[7] It is actually ambiguous whether **SS3** corresponds with **R5** or **R7**; the diagram matcher chooses one of the two possible correspondences arbitrarily.

correspond to objects that abut in the other diagram. For example, the algorithm constrains **B1**'s label to be **{E1,E2,E3}** — **B1** must correspond to an edge in Figure 30b that abuts the **Environment** since **B1** abuts the **Environment** and the **Environments** correspond in both diagrams. Since **SS3** abuts **B1**, this in turn places constraints on the correspondences of **SS3** — **R1** and **R3** are removed from its initial label of **{R1,R3,R5,R7}** because they do not abut any of the edges left in **B1**'s label. The diagram matcher also enforces the constraint that correspondences are one-to-one. In particular, if the label of an object **F** is reduced to the singleton set **{G}** then the algorithm constrains the label of **G** to be **{F}** and removes both **F** and **G** from the labels of all other objects.

The algorithm continues to constrain labels until no more information can be propagated. At this point, if some labels still contain more than one element, the algorithm selects one of the ambiguous labels and sets up separate matches for each choice of unique correspondence. After propagating constraints in Figure 30, for instance, **SS3** is labelled with both **R5** and **R7**, so two new matches are set up.

The algorithm continues propagating constraints and refining ambiguous labels until no more choices remain. At this point, the best match is chosen and returned. Two criteria are used to determine the best match. First, a match is considered better if it has fewer objects with empty labels, which signify objects that do not correspond to anything in the other diagram. Second, a geometric criterion is used if two matches have the same number of correspondences: the better match is taken to be the one with the most corresponding edges having the same orientation.[8] In Figure 30, the match in which **SS1** corresponds to **R1**, **SS2** to **R3**, and **B6** to **E12** is topologically as good as the match in which **SS1** corresponds to **R3**, **SS2** to **R1**, and **B6** to **E13**. In this case, the geometric criterion would prefer the match where **B6** and **E12** correspond, since their orientations are the same.

In matching the diagrams in Figure 30, the matcher reports that no objects in the goal diagram correspond with **R6**, **R7**, and all the edges connected to them. In general, the existence of such extra pieces is a bug that must be repaired by the debugger. An exception

[8] To compensate for noisy data, two edges are considered to have the same orientation if the angles are within 10%.

is where the extra piece falls outside the window of the goal diagram. For example, GORDIUS considers that the diagrams in Figure 31 match since the "extra" piece **IGN2** in the simulation diagram lies below the bottom edge of the goal diagram.

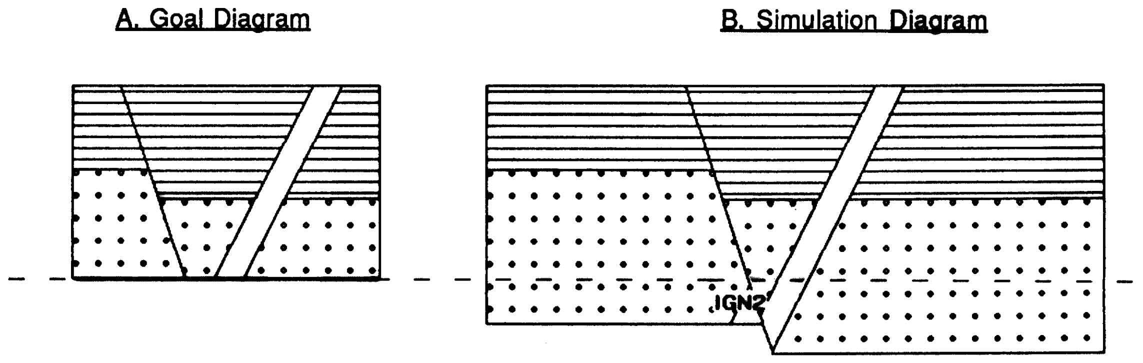


Figure 31. Successfully Matching Diagrams

The matching algorithm has several characteristics that make it well suited for doing geologic interpretation using the GTD paradigm. First, its primary reliance on topological matching corresponds well with our geologic knowledge that diagrams whose only differences are geometrical more nearly match since geometrical differences are generally easier to explain (e.g., via tilting) than topological differences.

Second, small topological differences in the diagrams are reported as small differences in the match, which corresponds well with our intuition that an hypothesis is close to being a solution if the simulation diagram closely matches the goal diagram. For example, the algorithm reports that all objects in Figure 30 have correspondences except for **R6**, **R7**, and their abutting edges. In comparing Figure 30a and Figure 31a, however, the matcher finds that few of the objects correspond, which is reasonable since the diagrams were formed from very different sequences of events.

Although the diagram matcher was developed specifically for doing geologic interpretation, it is fairly general and may be useful in other domains. In particular, the above discussion indicates that the domains should be ones in which topology is more important than geometry and the closeness of the match correlates with the closeness of the events that formed the diagrams.

3.3 Use of Simulation to Test Hypotheses

This section discusses how the causal and diagrammatic simulators implemented in GORDIUS combine to fulfill the roles of the tester in the GTD paradigm.

The tester has two major roles. First, it must verify whether an hypothesis actually solves the problem, that is, whether the hypothesized events can achieve the goal state from the initial state. To fulfill this role, the tester must be accurate — avoiding both false positives (i.e., accepting invalid solutions) and false negatives. The latter condition is especially important in order to prevent GTD from looping by continually rejecting legitimate solutions proposed by the generator or debugger.

The need for accuracy implies that the domain models and model of causality used by the tester must be at least as detailed as those used by the generator and debugger. In particular, while the generator can presume composability of events, the tester should make as few assumptions as possible about how events interact.

The tester's second role is to provide causal explanations for any bugs detected. As discussed in the next chapter, the causal explanations are crucial to the operation of the debugger. The more accurate and detailed the explanations, the greater the chance the debugger has of finding out what went wrong and fixing it.

Simulation is a natural technique for testing interpretations and plans. We can define simulation as the application of operators to a world model in order to predict the consequences of the operators. An operator is a function from world models to world models, where in GORDIUS a world model is a partial description of the world over an interval of time. Under this definition, testing a sequence of events is expressed very naturally as the application of operators (the events) to a world model (the initial state) to predict the consequences of the operators (the goal state).

Although simulation, in general, is well suited for testing sequences of events, the two types of simulations used in GORDIUS — causal and diagrammatic — have differences with respect to how well they fulfill the roles required of a GTD tester. The next two sections describe the strengths and weaknesses of the two simulation techniques in fulfilling those roles. We argue that the diagrammatic simulator is better at accurately verifying hypotheses, while the causal simulator is better in the role of providing causal explanations.

3.3.1 Accuracy in Simulation

In theory, any desired degree of accuracy can be obtained using causal simulation algorithms like the one described in this chapter. In practice, however, there are several difficulties in actually achieving the desired accuracy. While none are insurmountable, they all tend to make causal simulation more expensive to perform. As argued below, the desire for accurate simulations with tolerable computational costs is the impetus behind using diagrammatic simulation to determine complex spatial effects of geologic events.

One difficulty in achieving accuracy is providing the simulator with sufficiently detailed domain models. This, however, is not just a problem with causal simulation — any testing technique needs accurate models. A more severe difficulty is providing an adequate representation language for the domain models. For example, since discrete event models do not represent what happens during the occurrence of an event, our current simulator cannot reason accurately about simultaneous events that interact by affecting the same attribute at the same time. Although discrete models are sufficient for the domains we explored, extending the representation to use continuous process models (e.g., [Forbus]) might be necessary for other domains.

An interesting question is whether the potential accuracy of the causal simulator is limited by its use of local computation methods (i.e., determining the value of an expression solely from the values of its arguments). For example, it might be the case that certain degrees of accuracy can be obtained only by using more global methods, such as direct solution of simultaneous equations or iterative methods.

Although we do not yet have a complete answer, we believe that local methods are not an inherent limitation. The rationale is that since events in the real world act locally, we should be able to simulate their effects using local methods that mimic the causal nature of events. The problem is that relying on local methods tends to increase the cost of simulation. For example, we could accurately simulate deposition locally, particle by particle, but that would involve a ridiculous amount of computation.

The desire, then, is to achieve accuracy without sacrificing efficiency. These twin goals are achieved in GORDIUS by tailoring the tester to help overcome each of the three sources of difficulty listed above — modeling, representation language, and local computation methods. We argue that our causal representation language can be used to overcome the first difficulty, but that the other two suggest the use of a diagrammatic simulation technique tailored to the geologic domain.

The first step in obtaining efficiency in simulation is choosing a good modeling ontology — one that makes manifest the important aspects of the domain needed to solve the problem [Simmons & Davis, 83], [Van Baalen]. For example, the models used by both our causal and diagrammatic simulators explicitly represent those spatial objects and relations important in the geologic domain. The causal models represent rock-units, boundaries, and points; similarly, the diagrams represent faces, edges, and vertices. The causal models encode geometrical attributes, such as the height of a point and the orientation of a boundary; these same attributes are either encoded explicitly or easily computed in the diagram representation.

On the other hand, certain topological relations are represented only implicitly in the causal models, making them expensive to compute. For example, since the topology between a boundary and a rock-unit is encoded only by the **side-1** and **side-2** attributes of the boundary, the system must examine every boundary to find the set of boundaries surrounding a rock-unit. In contrast, the perimeter of a face is represented explicitly in the diagrams, making it trivial to determine all the edges surrounding a face. To make the causal simulator more efficient at simulating topological effects of events, we should update the model of rock-units to include a **perimeter** attribute.

Even with a good ontology, it may not be easy to simulate the effects of events accurately. For example, even with a **perimeter** attribute for rock-units, it is quite complicated to describe in our causal representation language what happens when a rock-unit splits in two — how new rock-units and boundaries are created as existing ones split, and how the new objects are related topologically to each other and to other existing objects.

The effects of events can be simulated more efficiently using representations (data structures and inference techniques) that are specialized for the types of changes that arise. This is the main impetus behind our use of a specialized diagrammatic representation — its associated manipulation techniques take advantage of topological connectedness to efficiently compute changes to the structure of the diagram. For example, the manipulation techniques that split a face do not examine its whole perimeter; instead they take advantage of the fact that the perimeter of the split pieces have much the same topology as the original face.

A major reason for the increased efficiency of the diagrammatic manipulation procedures is that they are operational descriptions for how to perform the simulation, not causal descriptions of what and why effects happen. These operational descriptions are most efficiently computed using non-local computation, such as by using global control constructs (sequencing and iteration) and by saving state during the computation. For example, the iterative line construction algorithm described in Section 3.2.3 computes how edges and faces split in $O(E^*P)$ time, where E is the total number of edges in the diagram and P is the number of edges intersected by the line. In contrast, it would take $O(E^3)$ time to determine the same effects using the local methods of our causal simulator.

In sum, the necessary degree of accuracy can be achieved with both the causal and diagrammatic simulators. The diagrammatic simulator, however, is more efficient due to its use of specialized representations (with their associated inference techniques) and non-local (procedural) methods of computation.

3.3.2 Producing Causal Explanations by Simulation

While major sources of inefficiency for the causal simulator stem from using local computation methods and describing effects causally rather than operationally, these are also the simulator's strength in its role of providing causal explanations.

By using local methods, in which the value of an expression depends only on the value of its arguments, dependencies can also be determined from the expression's arguments alone. By using causal descriptions that indicate why an effect happens, the dependencies recorded by the local methods will reflect the underlying causality of the description. In combination, the use of local methods and causal descriptions makes the production of causal dependency structures a natural by-product of our simulation algorithm.

Another advantage of using local methods is that evaluation methods may be added or modified without having to consider their effects on other methods. For example, the method to evaluate **Change** statements has gone through several modifications as our understanding of the underlying model of causality improved. In each case, the modification involved merely changing way **Change** statements were supported — the evaluation methods of other statements did not have to be modified.

On the other hand, the more global methods and operational descriptions used by the diagrammatic simulator are not amenable to producing adequate causal explanations. With the diagrammatic simulator, dependencies derived by tracing the simulation do not necessarily correspond to our intuitive notion of cause and effect. The problem is that based on such dependencies, the debugger would suggest repairs that are not causally related to fixing bugs. For example, in analyzing dependencies produced directly from simulating our line construction algorithm (Section 3.2.3), the debugger would suggest that adding faces above a given face might prevent it from being split. To us this repair seems silly, since we know that whether a line intersects a face is not causally related to the topology of its surrounding faces, even though it is easier to compute it that way.

To solve this problem of producing causal explanations, we do not record dependencies based directly on executing the diagrammatic procedures. Instead, only enough information is recorded by the diagrammatic simulator to enable a domain-dependent algorithm to construct *post hoc* causal explanations after testing is completed. For example, our line construction algorithm records which objects were split into pieces by which lines. If splitting an object **F** by line **L** turns out to be a bug, the system constructs an explanation for why the split occurred that depends on 1) the event for which **L** was constructed, 2) the existence of **F** at the time the event occurred, 3) the orientation and location of **L**, and 3) the height, width, and location of **F**.

This approach of separating the simulation algorithm from the mechanism used to create explanations has the advantage that the tester can use non-causal simulators, which are often easier to develop and more efficient to run, while it can still produce the causal explanations needed by the debugger. Care must be taken, however, to ensure that the explanations produced accurately reflect the results of the non-causal simulation.

An additional problem with non-causal testing methods is that it is often more difficult to determine which goals are not achieved. An extreme case, for instance, is a tester that computes a single statistic and compares it to some threshold, since this reveals almost no information about what went wrong when the test fails. In particular, since that is the case with the tester used by DENDRAL, debugging is not a viable option in that domain [Buchanan, personal communication]. In our own domain of geologic interpretation, it took some effort to derive a diagram matching algorithm that can determine which objects are and are not in correspondence, information essential in determining which goals are achieved by an hypothesis.

In conclusion, there are tradeoffs between causal and diagrammatic simulators in their roles as testers. The diagrammatic simulator is better in the role of verifying hypotheses accurately since it can simulate detailed domain models efficiently by using operational descriptions of events and global computation methods. The causal simulator is better at providing explanations since its local methods and causal event descriptions facilitate recording dependencies that correspond to the causal nature of events.

Our approach of combining both causal and diagrammatic simulations adheres to the general philosophy of GTD that more detailed knowledge should be brought to bear only if and when less detailed knowledge fails to solve the problem. The causal simulation in effect forms a skeleton of the effects of the geologic events, and the diagrammatic simulation allows us to flesh it out.

4. Debug

The task of the debugger is to modify hypotheses to repair the bugs detected by the tester. Our approach to 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. If the predicted state of the world does not match the desired state, it must be that one of the underlying assumptions is faulty and needs to be replaced.

Our debugging approach uses three causal reasoning techniques to help focus on which assumptions to replace and how to replace them. First, the debugger analyzes causal dependency structures to locate the assumptions upon which the bugs depend. Second, the debugger regresses values back through dependency paths to indicate the direction in which to change the assumptions. Third, domain-independent repair strategies are used to suggest ways to replace assumptions in order to fix a bug. In addition, an evaluation heuristic uses causal reasoning to estimate the global effects of each suggested repair, determining whether it adds new bugs and/or serendipitously repairs remaining bugs.

We refer to our debugger as *assumption-oriented* since it works by replacing the faulty assumptions underlying a bug. Technically, an assumption is defined as a statement whose value does not depend on any other statements. Both propositions and expressions can be assumptions. For instance, the system can assume that the proposition **Occurs(Erosion, Erosion1)** is true, and it can assume that the value of the expression **Elevel1** is 50.

A bug is defined as an inconsistency between the desired value of a statement and its predicted value. The tester detects bugs that arise when 1) a desired event cannot occur because its preconditions are not met, 2) the desired effects of events do not occur because parameter constraints do not hold, and 3) a goal is predicted to be unachieved in the final simulated state. In each case, the tester constructs two causal dependency structures — one explaining how the predicted value arises and the other explaining why the desired value is needed (the latter typically consists of a single assumption, for instance, the assumption that some event is supposed to occur).

The debugger begins repairing a bug by tracing back through the two dependency structures explaining the predicted and desired values to locate the assumptions upon which the bug depends. Using both dependency structures gives the debugger an added degree of flexibility — it can repair a bug either by changing the predicted value to match the desired value, or by changing the desired to match the predicted.

To help determine the direction in which to change an assumption, the desired value of the bug is regressed back through the dependency structure explaining the predicted value (and *vice versa*). For example, suppose that the predicted value of **R1.orientation@Plan-end** depends on the constraint:

R1.orientation@Plan-end = Theta2 + 3°,

where **Theta2** is the parameter of a tilt event. If we desire the value of **R1.orientation@Plan-end** to be 10°, regression indicates that changing the value of **Theta2** to (10-3)° will repair the bug.

The debugger uses six domain-independent repair strategies to determine how to replace assumptions. The repair strategies reason about the type of assumption being replaced, the regressed values, and the event models of the domain. The strategies suggest ways to repair a bug by adding or deleting events, changing parameter bindings, or changing orderings between events. For example, the repair strategy for a bug that depends on an assumption about the value of a parameter is to replace the parameter value with the one obtained through regression. Similarly, a repair strategy for an assumption that some attribute persists in value during an interval is to add a new event during the interval that can affect the attribute.

Our debugging methodology typically suggests many repairs for each bug. To help control the search for a solution, the debugger uses an evaluation heuristic to estimate the global effects of the suggested repairs, and pursues the one estimated to come closest to solving the overall problem. The debugger continues until it estimates that all bugs are repaired, at which point the repaired hypothesis is submitted to the tester for verification.

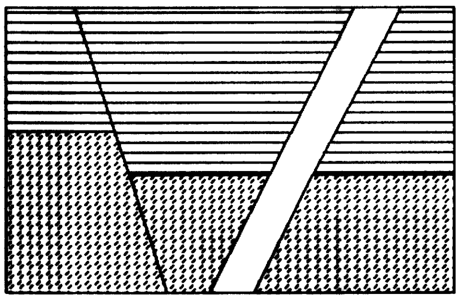
The evaluation heuristic used by the debugger is based on the tester's causal simulator. It extends the causal simulator by handling partially ordered hypotheses and by incrementally updating causal dependency structures based on modifications to hypotheses.

For efficiency reasons, however, our evaluation heuristic is not complete, occasionally failing to predict accurately which bugs remain.

The next section illustrates GORDIUS' debugging capabilities on a simple example. Section 4.2 describes how the debugger locates assumptions and regresses values, and details the six repair strategies currently implemented in GORDIUS, and discusses the evaluation heuristic used to control search. Section 4.3 analyzes the completeness and applicability of our theory of debugging. In particular, we argue that the combination of dependency tracing, regression, and our set of repair strategies is sufficient to handle a wide range of bug types arising from many different combinations of assumptions.

4.1 Debugging an Incorrect Hypothesis

Continuing our example from Sections 2.1 and 3.1, we examine how the debugger handles the bug that arises in testing the interpretation of Figure 32 — that erosion does not occur because its precondition that the surface of the Earth is below sea-level does not hold.



1. Deposition1(ROCK1, Shale)
2. Batholithic-Intrusion2(ROCK2, Granite)
3. Faulting1(FAULT1)
4. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
5. Erosion1(EROBOUND, Elevel1)

Figure 32. An Incorrect Interpretation of the Diagram

Figure 33 shows the two causal dependency structures produced for this bug — one is the trivial explanation that the generator assumed that erosion occurred; the other explains why the causal simulator predicts that the surface of the Earth would be below sea-level at the start of the erosion event.[1] This prediction depends on

[1] Figure 33 shows only part of the dependency structure for the predicted value — the complete structure produced by GORDIUS has about twice as many nodes.

knowledge about the value of sea-level and the value of the height of the top of the Earth's surface. The predicted height of the surface, in turn, depends on various factors, including that it persisted in value from the end of shale deposition to the start of erosion, and that it increased by **Dlevel1** due to the effects of the deposition of shale.

The debugger locates the assumptions underlying the bug by tracing back to the leaf nodes of the dependency structures (the boxed statements in Figure 33). At the same time, the debugger regresses values back through the dependencies to indicate the direction in which to change the assumptions. For example, the regression indicates that **Surface.top.height@Erosion1.start** should be greater than **Sea-Level** in order to get erosion to occur. Regressing further indicates that the desired value of **Dlevel1**, the amount of shale deposited, is constrained by the expression:

Dlevel1 > (Sea-Level - Surface.top.height@Deposition1.start),

that is, enough deposition should be done to raise the surface above sea-level.

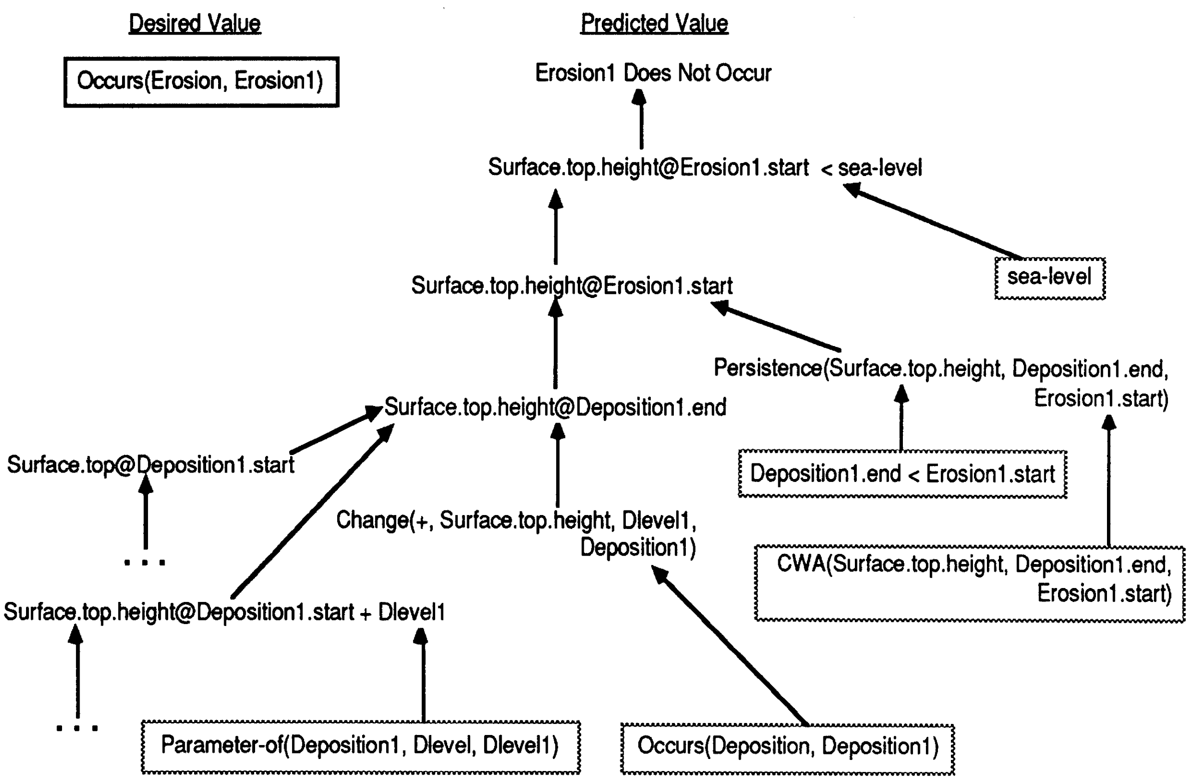

Desired Value
Predicted Value
Occurs(Erosion, Erosion1)
Erosion1 Does Not Occur
Surface.top.height@Erosion1.start < sea-level
sea-level
Surface.top.height@Erosion1.start
Persistence(Surface.top.height, Deposition1.end, Erosion1.start)
Surface.top.height@Deposition1.end
Surface.top@Deposition1.start
Deposition1.end < Erosion1.start
Change(+, Surface.top.height, Dlevel1, Deposition1)
CWA(Surface.top.height, Deposition1.end, Erosion1.start)
Surface.top.height@Deposition1.start + Dlevel1
Parameter-of(Deposition1, Dlevel, Dlevel1)
Occurs(Deposition, Deposition1)

Figure 33. Partial Causal Dependency Structure

The debugger uses its repair strategies to consider each of the assumptions illustrated in Figure 33:

1. **Occurs(Erosion, Erosion1)** (an erosion event occurs) — One way to repair the bug is to change the intention that erosion is supposed to occur. The debugger has two general strategies for accomplishing this — 1) delete the event altogether, or 2) replace it with a similar event that achieves the same goals but has different preconditions. Following the first strategy, the debugger proposes deleting the erosion event; for the second strategy, it finds no way to replace the event, since it knows of no other event that can remove parts of the shale and mafic-igneous formations.

2. **Sea-Level** (sea-level has a particular value) — This assumption is not handled because the debugger presumes that the values of constants are unchangeable.

3. **Deposition1.end < Erosion1.start** (the deposition event occurs before erosion) — The general strategy is to reorder the events. In this case, however, the debugger determines that putting erosion before deposition will not help since the precondition of erosion will still not be met — the surface was even further below sea-level before the deposition occurred.

4. **CWA(Surface.top.height, Deposition1.end, Erosion1.start)** (the closed-world assumption that nothing changes the height of the top of the surface between deposition and erosion) — The repair strategy for such persistence assumptions is to add an event that can affect the attribute. In this case, that means inserting an event between deposition and erosion that can change the height of the surface. The regressed value of **Surface.top.height@Erosion1.start** indicates that the event needs to increase the height of the surface to a value greater than **Sea-Level**.

   There are four types of events in our domain models that can affect the height of the Earth's surface — faulting, subsidence, tilt, and uplift. The debugger rejects faulting and subsidence because it determines that their effects are to decrease height. Adding uplift is suggested as a possible repair because its effect, raising the height of all geologic objects, is what is needed. Adding a tilt event is also suggested. Even though tilt can either increase or decrease height, depending on its direction and origin, the debugger considers the resulting uncertain situation to be an improvement over the currently known, but buggy, one.

5. **Occurs(Deposition, Deposition1)** (a deposition event occurs) — As with #1 above, the two repair strategies are to delete or replace the event. In this case, both strategies fail. The debugger determines that deleting **Deposition1** would not repair the bug since the surface of the Earth would still be below sea-level. The event cannot be replaced because the debugger fails to find an alternative event that can create shale.

6. **Parameter-of(Deposition1, Dlevel, Dlevel1)** (the amount of deposition done is **Dlevel1**) — The strategy here is to change the parameter to a value that fixes the bug. As described above, the regression indicates that the desired value for **Dlevel1** is:
   Dlevel1 > Sea-Level - Surface.top.height@Deposition1.start.
   This value, however, conflicts with a constraint in our model of deposition that no amount of deposition can raise the Earth's surface above sea-level. Thus, the debugger concludes that **Dlevel1** cannot be changed in such a way as to repair the bug.

Other strategies considered for parts of the causal dependency structure not shown include increasing the height of the surface before the deposition event occurs, and changing the geologic top of the surface (e.g., by depositing another formation on top of the shale). Since none of these other strategies succeed in this example, the debugger suggests a total of three repairs for the bug that erosion does not occur — deleting the erosion event, inserting an uplift event, and inserting a tilt event.

The debugger evaluates each of the suggested repairs to estimate the total number of bugs remaining after each repair is done. The evaluation heuristic determines that the uplift event fixes the bug and introduces no new bugs, but that the other two repairs introduce new bugs. In particular, if erosion is deleted, the surface will not end up being horizontal; if tilt is added, the orientations of the boundaries and rock-units will no longer match their orientations in the goal diagram.

Since the debugger prefers hypotheses that have fewer remaining bugs, it modifies the initial hypothesis to insert an uplift event between the end of deposition and the start of erosion (Figure 34).

From the results of the regression, the amount of uplift (**Uamount1**) is constrained to be enough to raise the surface above sea-level. The modified hypothesis is verified by the tester, which concludes that the hypothesis (or, at least, one of its linearizations) is in fact a plausible interpretation of the diagram in Figure 32.

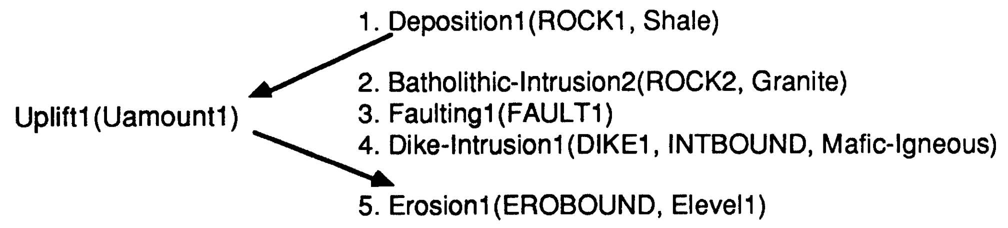

1. Deposition1(ROCK1, Shale)
2. Batholithic-Intrusion2(ROCK2, Granite)
3. Faulting1(FAULT1)
4. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
5. Erosion1(EROBOUND, Elevel1)
Uplift1(Uamount1)

Figure 34. The Modified Hypothesis with Uplift.

4.2 The Debugging Algorithm

The next three sections describe the reasoning techniques used by our debugging algorithm — 1) locating underlying assumptions and regressing values through dependency structures, 2) using repair strategies to suggest ways to replace assumptions, and 3) evaluating the suggested repairs to determine how they affect achievement of the other goals of the problem.

4.2.1 Locating Underlying Assumptions and Regressing Values

The debugger focuses on potentially faulty assumptions by analyzing causal dependency structures. Dependency structures are represented using a TMS [McAllester], where the belief that a statement has a particular value is represented by a TMS node, and the causal support for that belief is represented by *justifications*. For an assumption, a justification consists of a symbol that represents the degree of belief (e.g. **given** or **closed-world**); for a non-assumption node, a justification is a set of TMS nodes plus the inference rule used to derive the node's value from its support nodes. Nodes can have multiple justifications. For instance, the belief that an event occurs may be supported both by an assumption made by the generator and by the belief that its preconditions hold.

Locating the assumptions underlying a bug is relatively simple. The debugger traces through all justifications of a node back to the assumption nodes — those that have at least one justification with no supporting nodes. Each path to an assumption is collected since the different paths represent alternative explanations for why the bug depends on the assumption.

The debugger eliminates a path from consideration if its underlying assumption has been made with a very strong degree of belief. An assumption's degree of belief is represented using a partially ordered set of symbols. Those above a certain threshold (**given, constant, domain-model,** and **physical-truth**) are considered too strongly held to be changeable by GORDIUS. In essence, GORDIUS treats assumptions of this class as premises.

While the debugger traces dependencies, it also regresses node values back through the justifications. In essence, the regression inverts the dependence between a statement and its supports, indicating how the values of supporting nodes should be changed to achieve the desired value of the dependent node.

Associated with each type of statement in our representation language is a function that takes a symbolic constraint on the value of a statement and returns constraints on the values of its supports. For example, if the regressed value of **(A and B)** is **true**, the function associated with the **and** predicate determines that the regressed values of both **A** and **B** are **true**. Similarly, if the regressed value of the expression **X-Y** is **3**, the system infers that the regressed value of **X** is the constraint **X=Y+3**, and the regressed value of **Y** is **Y=X-3**.

Figure 35 shows the regressed values (italicized) for one of the dependency paths underlying the bug in Section 4.1 that erosion does not occur. The regressed values indicate that the bug can be fixed by negating the **CWA** assumption (a closed-world assumption that nothing changes the height of the top of the Earth's surface) and replacing it in a way that achieves the constraint:

Surface.top.height@Erosion1.start > Sea-Level.

The debugger algebraically simplifies the regressed constraints to facilitate subsequent reasoning about them. For example, regressing the constraint **X > (Z + W)** back through the dependency **X = Y + Z** yields the constraint **Y > (Z + W) - Z**, which is simplified to **Y > W**.

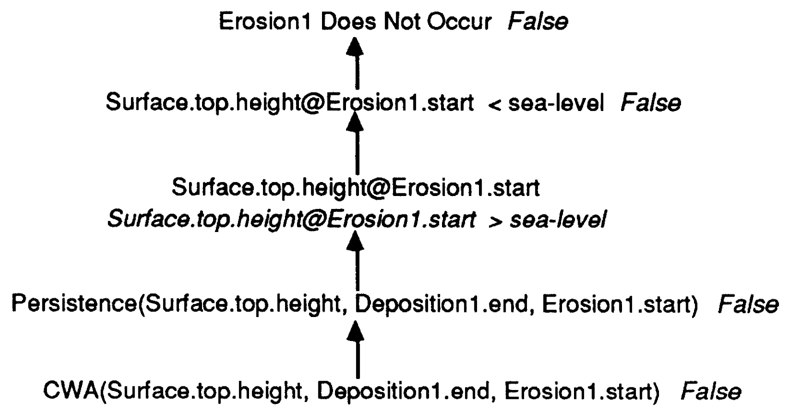

Erosion1 Does Not Occur *False*
Surface.top.height@Erosion1.start < sea-level *False*
Surface.top.height@Erosion1.start
Surface.top.height@Erosion1.start > sea-level
Persistence(Surface.top.height, Deposition1.end, Erosion1.start) *False*
CWA(Surface.top.height, Deposition1.end, Erosion1.start) *False*

Figure 35. Regression of Desired Values (Italicized) Through a Dependency Path

For user-defined functions that do not have an associated definition (e.g., **Foo(x, y)**), the debugger creates new functions for each argument in the function's domain (e.g., **Foo$_x$-inverse** and **Foo$_y$-inverse**) and uses them to do the regression. For example, regressing **1** through **Foo(a,b)** produces **a=Foo$_x$-inverse(1,b)** and **b=Foo$_y$-inverse(1,a)**. Although this trick enables the debugger to regress values, it does not provide much leverage in trying to select appropriate repairs. For instance, it does not indicate how **a** can be changed in order to make **Foo(a,b)** equal to **1**.[2]

Since the dependency structures underlying bugs tend to be quite large, the debugger employs techniques to prune the search for underlying assumptions. The three pruning methods used all look for situations in which a node cannot be changed in any way to achieve its regressed value, which indicates that its underlying assumptions cannot be changed in any way to repair the bug. For example, if **a = b/c**, there is no way to change **c** to achieve the constraint **a = 0** (assuming **c** is finite).

One pruning method stops tracing at a node if the regressed constraint does not contain the name of the node itself. This situation indicates that the regressed constraint is independent of the node, so changing the node's value will have no impact on fixing the bug. For example, Figure 36b illustrates a path in the dependency structure derived from simulating the hypothesis of

[2] Section 5.4 illustrates how such inverse functions do provide useful information for the diagnosis of faults in semiconductor fabrication.

Figure 36a. The italicized statements are the constraints obtained by regressing the desired value that **R1.orientation@Plan-end** should be greater than **R1.orientation@Tilt1.start**. The regression is pruned at the node **R1.orientation@Tilt1.start** since its name does not appear in the regressed constraint **Theta1 > 0**. This indicates that no change to **R1**'s orientation at **Tilt1.start** will affect its relationship with **R1.orientation@Plan-end**.

<u>A. Current Hypothesis</u>

1. Deposition1(R1, Shale)
2. Tilt1(Theta1)

<u>B. Dependency Path and Regressed Values (*Italicized*)</u>

R1.orientation@Plan-end
R1.orientation@Plan-end > R1.orientation@Tilt1.start

↑

R1.orientation@Tilt1.end
R1.orientation@Tilt1.end > R1.orientation@Tilt1.start

↑

R1.orientation@Tilt1.start + Theta1
R1.orientation@Tilt1.start + Theta1 > R1.orientation@Tilt1.start
simplifies to: Theta1 > 0

↑

R1.orientation@Tilt1.start

Figure 36. Pruning the Trace Through Dependency Structures

A related pruning method stops tracing if the regressed value of a node is inconsistent, most notably, if it falls outside its legal range of values. For example, in regressing the desired value:

R1.orientation@Plan-end = R1.orientation@Tilt1.start

through the dependency path in Figure 36b, the regressed value of **R1.orientation@Tilt1.start + Theta1** is found to be **Theta1 = 0**. This conflicts with our tilt event model which constrains its **Theta** parameter to be non-zero. Thus, it is futile to replace any of the assumptions underlying this statement, since any change that achieves the desired value would be inconsistent with the domain models.

The third pruning method is applicable when the debugger regresses the value **false** through a conditional statement of the form:

(If antecedent Change(type, object.attribute, magnitude, event)),

indicating that the bug might be repaired if the change did not happen. In this case, the debugger traces back through **antecedent** only if the predicted value of **object.attribute@event.start** is closer to the regressed value of **object.attribute@event.end** than is its current predicted value. This reflects the observation that it is useless to try to prevent a change from happening if the situation will be no better without the change's effect.

As the debugger traces dependencies and regresses values, it applies the three pruning methods to each node, halting the search through a node if any pruning method is found to be applicable. Our experience shows that, in combination, the three pruning methods significantly reduce the number of assumptions that would otherwise have to be considered by the debugger's repair strategies.

4.2.2 Repairing Bugs

For each dependency path collected, the debugger suggests ways to replace the assumption at the end of the path with ones that will achieve the desired value of the statement at the head of the path.

We have implemented bug repair strategies for the six different types of assumptions that our experience and analysis show are responsible for most of the bugs in a wide variety of domains. The strategies can replace faulty assumptions about 1) which events occur, 2) parameter bindings of events, and 3) temporal orderings between events, as well as three types of closed-world assumptions: 4) the assumption that an attribute persists in value because no known event affects it, 5) the assumption that an object continues to exist because no known event destroys it, and 6) the assumption that the system knows about all objects of a given type.

The repair strategies all share a common framework in that they modify the current hypothesis by retracting one underlying assumption and adding assumptions about new event occurrences, parameter bindings, and/or temporal orderings. The strategies all determine how to repair bugs by analyzing the dependency paths, the regressed values along the paths, and the domain's event models.

4.2.2.1 Producing Occurrence Repairs

Several of the repair strategies, plus parts of the evaluation heuristic described in Section 4.2.3, need to determine which events can affect some temporal reference in order to achieve a given value. For example, in trying to repair the bug in Section 4.1 that erosion does not occur, the debugger needs to find events that can change the value of **Surface.top.height@Erosion1.start** to be greater than sea-level.

The technique used to determine the set of such events takes as input 1) a temporal reference of the form **object.attribute@time**, 2) a symbolic constraint on the desired value of the temporal reference, and 3) an initial list of events that might affect the **object**'s **attribute**. The technique produces a list of *occurrence repairs*, which are events that, if inserted into the current hypothesis, can achieve the desired value.

The technique works by determining, for each event in the initial list, whether there exist parameter-binding constraints that can be added to the event so that it can achieve the desired value. If more than one set of such binding constraints exists, (i.e., the event can achieve the value in more than one way), multiple occurrence repairs are created for that event, each having different parameter constraints.

An event can possibly achieve the desired value of a temporal reference if: 1) it contains a change statement that can affect the **attribute** of the temporal reference, 2) the change statement can affect the **object** of the temporal reference, 3) all conditions associated with the change statement are potentially achievable, and 4) the magnitude of the effect can change the temporal reference to the desired value.

The first step is to determine whether any change statement of the event can affect the **attribute** of the temporal reference. This step is efficient in GORDIUS because change statements of events are indexed under the attributes that they affect, so finding the relevant effects is a simple look-up operation. For example, associated with the **top** attribute are two effects of the blocks-world **Puton** event:

```
1) Change(=, Dest.top, {Source}, Puton) and
2) (If (Source ∈ $b.top@Puton.start) and ($b ≠ Dest)
        Change(-, $b.top, Source, Puton)),
```

where the first effect stacks the **Source** block on the **Dest** block and the second effect removes the **Source** block from the top of all blocks it is currently on (except for the one onto which it is being moved).

The second step of the technique determines whether the change statements found can affect the **object** of the temporal reference. If the object is a constant or bound process parameter, the **object** of the temporal reference must be the same constant; if the object of the change statement is a quantified variable or unbound parameter, the **object** must be a compatible type. In the latter case, constraints are added to the event indicating that the variable or parameter is bound to the **object**. For example, given the temporal reference **A.top@Plan-end**, the technique produces two occurrence repairs for **Puton** (using the change statements above) — one with the **Source** parameter bound to **A**, and the other with the variable **$b** bound to **A**.

An additional condition in matching parameters is that the **object** must satisfy any parameter constraints of the event (i.e., propositions in the event's constraints field that consist only of parameters, constants, and function or relation symbols). For example, the **Elevel** parameter in the erosion event is constrained by the proposition **Elevel** ≥ **Sea-Level**. In addition, parameters representing objects created by the event have the implicit constraint that the object must not already have been created by another event. This, for instance, rules out suggesting that the change statement **Change(=, Boundary.orientation, 0, Erosion)**, where **Boundary** is the created erosional boundary, can be used to alter the orientation of a pre-existing boundary.

The third step of the technique is to determine whether all of change statement's conditions can be achieved. An achievable condition is one that either currently holds or can potentially be achieved by subsequent debugging. We use a simple heuristic to recognize potentially achievable conditions — if the condition has terms denoting time points or intervals, it is presumed that some repair strategy can change the appropriate temporal references needed to make the condition hold. For example, the condition **(Source** ∈ **$b.top@Puton.start)** is considered achievable — even if it does not currently hold there are repair strategies that can achieve it by changing the value of **Source.top** to include block **$b**. On the other hand, if the condition **($b** ≠ **Dest)** in change statement #2 above does not hold, it is considered unachievable. In other words, it is unacceptable for the debugger to propose removing block **A** from the top of block **C** by putting **A** back on **C**).

The final step in producing occurrence repairs is to determine whether the magnitudes of the effects found are sufficient to achieve the desired value of the temporal reference. For a magnitude that is a simple expression (i.e., a constant, variable, parameter, or temporal reference), it is sufficient if the magnitude matches the desired value in the manner described above — constants, bound variables, and parameters must be the same as the desired value; free variables and parameters must have the same type as the desired value. For the temporal reference **A.top@Plan-end**, for instance, the magnitude of the effect:

Change(=, Dest.top, {Source}, Puton)

is sufficient to achieve the desired value **{B}** with the added constraint that the **Source** block be bound to **B**.

A more semantic approach is necessary for magnitudes that are expressed in terms of functions. In such cases, the debugger uses its knowledge of arithmetic, sets, and the meaning of the **Change** statement to determine whether any consistent bindings can be found for free variables or parameters in the magnitude. The debugger creates an equation relating the magnitude of the change statement and the desired value, and symbolically solves for the free parameters and variables in the equation. The resulting expressions are then evaluated to determine whether they are consistent with the domain model's constraints on the parameters and variables.

For example, to decide whether the statement **Change(-, $b.top, Source, Puton))** is sufficient to achieve the value **{B}** for **A.top@Plan-end**, the debugger binds **A** to the variable **$b** and creates the equation:

{B} = A.top@Puton.start - {Source}.

Solving for **Source**, the debugger produces:

Source ∈ A.top@Puton.start - {B}.

Using the current predicted value of **A.top@Puton.start**, the debugger evaluates the set-difference expression. The magnitude is sufficient to achieve the desired value if the resultant set contains exactly one element — in which case the debugger constrains the parameter binding for **Source** to be that element.

Similarly, to determine whether the magnitude of **Change(+, $pt.height, Uamount, Uplift)** can achieve the desired value that **Surface.top.height@Erosion1.start** be greater than **sea-level**, the debugger chains together the three equations:

Surface.top.height@Erosion1.start > Sea-Level
Surface.top.height@Erosion1.start = Surface.top.height@Uplift.end
Surface.top.height@Uplift.end = Surface.top.height@Uplift.start + Uamount

and solves the result for **Uamount**, producing the expression:

Uamount > Sea-Level - Surface.top.height@Uplift1.start.

Since **Uamount** is constrained by our domain models to be positive, the magnitude is sufficient if the arithmetic difference evaluates to a positive quantity, indicating that there is some amount of uplift that can achieve the desired value. In such a case, the debugger would create an occurrence repair for an uplift event, adding the inequality above as a parameter constraint on **Uamount**.

Sometimes the desired value cannot be achieved in one debugging step. To handle such cases, the magnitude of an effect is considered sufficient if it moves the value of the temporal reference closer to the desired value. A reasonable definition of "closer" is to use less-than for quantities, and subset for sets (e.g., set **B** is closer to **A** than is **C** if $\mathbf{A} \subseteq \mathbf{B} \subseteq \mathbf{C}$ or $\mathbf{C} \subseteq \mathbf{B} \subseteq \mathbf{A}$). In fact, our debugger uses an even more conservative approach — the magnitude of an effect is considered sufficient as long as it does not move the temporal reference away from its desired value. For example, the debugger suggests adding tilt to increase the height of the Earth's surface, even though tilt may either increase or decrease height depending on the tilt's angle and origin, neither of which may be known exactly.

Applying these four steps to all events in the initial input list of events produces a list of occurrence repairs that are sufficient for achieving the desired value of the given temporal reference. As mentioned, these occurrence repairs form the basis for several of the bug repair strategies described in the following sections.

4.2.2.2 Replacing Event-Occurrence Assumptions

An event-occurrence assumption — **Occurs(type, event)** — indicates that an **event** of **type** is assumed to occur. Both the generator and debugger can make such assumptions, the only difference being that the debugger makes them with a higher degree of belief since it analyzes more carefully whether the event actually achieves the desired goals.

One strategy for handling an event-occurrence assumption is simply to delete the event from the hypothesis. The situations in which this strategy is applicable differ for the two ways in which such assumptions can appear in our causal dependency structures. The first situation is where the event is assumed to occur but currently cannot because some of its preconditions do not hold. For this case, the deletion strategy is always applicable.

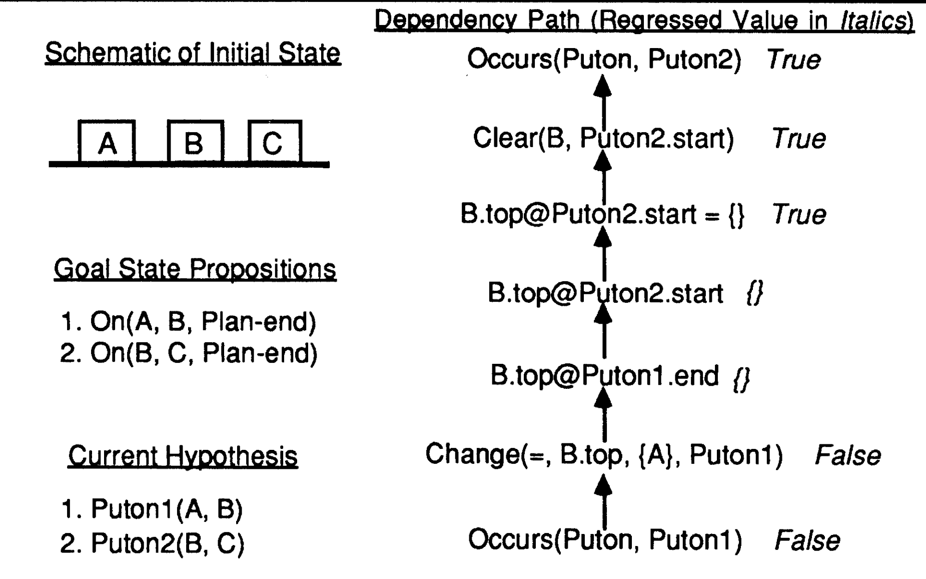
The second situation is where an effect of the event contributes to the bug by changing the value of some temporal reference. This *buggy temporal reference* can be identified by tracing forward from the assumption to find the first temporal reference along the dependency path being analyzed. For the assumption **Occurs(Puton, Puton1)** in Figure 37, for example, the buggy temporal reference for is **B.top@Puton1.end**.

In such situations, the deletion strategy is applicable only if the desired value of the buggy temporal reference will be achieved (or, at least, will be closer) if the event does not occur. Operationally, GORDIUS determines the effect of deleting an event (**E1**) on a temporal reference (**obj.attr@E1.end**) by evaluating its value at the start of the event (**obj.attr@E1.start**). In the problem of Section 4.1, for instance, the bug that erosion does not occur depends in part on the value of the height of the Earth's surface, which in turn is affected by the occurrence of shale deposition. Deleting the deposition event is not applicable in this case, however, because the height of the surface was even further below sea-level before the deposition occurred.

Deleting an event is a fairly drastic type of repair. It often introduces more bugs than it fixes since the system include events to satisfy certain goals and it is likely that these goals would not be achieved if the events were deleted. A more useful repair strategy for handling event-occurrence assumptions is to replace the event with a similar event.

An event is taken to be "similar" if it has the same important effects as the original event but avoids the problem that led to the bug. For GORDIUS, the important effects of an event are not pre-stored, but are determined dynamically by finding the effects that contribute to achieving the problem's goals. For example, the important effect of a **Puton** event may be either the one that stacks two blocks or the one that clears one block off another, depending on the problem being solved.

In addition to having the same important effects, a similar event is defined as one that avoids the bug. For cases in which a precondition of the event does not hold, similar events must not have the offending precondition. For cases in which an effect of the event supports the bug, similar events must achieve the desired value of the buggy temporal reference. In Figure 37, for example, **Puton1** is hypothesized to achieve the goal of having **A** on **B**, and **Puton2** is used to achieve the goal of **B** on **C**. The bug is that **Puton2** cannot occur because **Puton1** interferes with its precondition that **B** be clear.[3] The **Puton2** event can be replaced with an event that puts **B** on **C** but does not require **B** to be clear. In this problem, **Puton1** cannot be replaced because its important effect is the same as the one leading to the bug — any event that achieves the goal of having **A** on **B** cannot possibly clear **B** at the same time.


Schematic of Initial State
A
B
C
Goal State Propositions
1. On(A, B, Plan-end)
2. On(B, C, Plan-end)
Current Hypothesis
1. Puton1(A, B)
2. Puton2(B, C)
Dependency Path (Regressed Value in *Italics*)
Occurs(Puton, Puton2) *True*
Clear(B, Puton2.start) *True*
B.top@Puton2.start = {} *True*
B.top@Puton2.start *{}*
B.top@Puton1.end *{}*
Change(=, B.top, {A}, Puton1) *False*
Occurs(Puton, Puton1) *False*

Bug: **Puton2** cannot currently occur because a precondition does not hold

Figure 37. Replacing the Occurrence of an Event

The algorithm for replacing an occurrence by a similar event begins by collecting the important effects of the occurrence. The important effects of an event **E1** are found by tracing forward along all paths in the dependency structure from the event-occurrence assumption to the top-level goals, then regressing the desired

[3] This example, like many of those this chapter, is contrived to exhibit the range of our debugger. In solving this problem, GORDIUS would actually find a correct solution before proposing the buggy hypothesis of Figure 37.

values of the goals through those paths back to temporal references of the form **obj.attr@E1.end**.

These temporal references are used to construct the set of similar events. The debugger first chooses one of the temporal references and produces an initial set of occurrence repairs that achieve the regressed value of the temporal reference.[4] That set of events plus another of the temporal references are used to produce another, more constrained, set of occurrence repairs. The algorithm continues to iterate until either a step produces no occurrence repairs, in which case there are no similar events, or all the temporal references have been processed, in which case the output of the final iteration consists of events that achieve all the goals achieved by the original event.

Finally, the debugger removes those occurrence repairs that do not avoid the bug. For the case in which the original event has an unachieved precondition, an occurrence repair is removed if it has the same precondition. For the case in which an effect of the original event supports the bug, one more iteration is performed using the buggy temporal reference to yield a set of occurrence repairs that repair the bug as well as achieving all the goals of the event being replaced.

Note that our definition of "similar" events is fairly conservative — a similar event must achieve *all* the important effects of the original event. We have chosen this definition to limit the number of replacements suggested by the debugger. This definition of similarity, however, does not impinge on the completeness of the debugger since in cases where it is too strict, the desired debugging can be accomplished using combinations of the other repair strategies. Instead of replacing an event, for instance, the debugger could delete it and subsequently insert several other events which in concert reachieve the goals supported by the deleted event.

4.2.2.3 Replacing Parameter-Binding Assumptions

The parameter-binding assumption — **Parameter-of(event, formal, actual)** — indicates that **actual** is the binding of the

[4] The initial list of events used for producing occurrence repairs consists of one event for each type known to the system, except for the type of the event being replaced. The constraints on their temporal orderings are the same as those of the replaced event.

formal parameter for the **event**. Such assumptions are made by the generator in instantiating local interpretations, and by the debugger in producing occurrence repairs. In addition, if parameters are still unbound when an hypothesis is ready to be tested, the tester will choose arbitrary bindings that are consistent with the existing parameter constraints. For example, if the hypothesis were "put block **A** somewhere," the tester would choose some object as the destination of the **Puton** event, subject to the constraint that it not be block **A** itself.

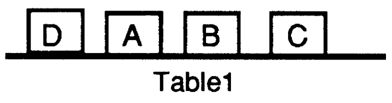
The repair strategy for parameter-binding assumptions is to find alternative bindings for the formal parameter that will repair the bug. Constraints on the bindings are found by regressing the desired value of the bug back through the dependency path. The regressed constraint is conjoined with the constraint that the new parameter value not equal its old value, plus any parameter constraints for the **event**. This produces a *parameter-value* constraint on the desired binding for the **formal** parameter.

The debugger then creates occurrence repairs having bindings consistent with the parameter-value constraint. If the type of the formal parameter is real-valued, the debugger creates a new quantity and asserts that the parameter-value constraint holds for that quantity. For all other parameter types, separate occurrence repairs are created for each existing object that is both of that type and consistent with the parameter-value constraint.

For example, the bug in Figure 38 is that **Puton2** cannot occur because **D** is not clear. Regressing through the dependency structure the desired value that **Clear(D, Puton2.start)** be true produces the constraint **Source.top@Puton2.start={}**. The debugger conjoins this with the constraint not to reuse the old parameter value and the parameter constraint of the **Puton** event that a block cannot be put on itself, producing a constraint on the desired binding of **Source** for **Puton2**:

Source.top@Puton2.start={} and Source≠D and Source≠Dest.

Of the existing blocks in Figure 38, only **C** and **E** are consistent with this constraint. While debugger suggests both as replacements for **D**, the evaluation heuristic determines that block **E** is a better choice since binding **Source** to **C** introduces the new bug that **On(C, D, Plan-end)** is no longer achieved.

Initial State	Goal State	Current, Buggy Hypothesis
	1. On(C, D, Plan-end) 2. Covered(A, Plan-end)	1. Puton1(C, D) 2. Puton2(D, A)

<u>Bug</u>: **Puton2** cannot occur because **D** is not clear at the start of **Puton2**

Figure 38. Replacing a Parameter-Binding Assumption

Another example of the applicability of this repair strategy is shown in Figure 39. The bugs are that the orientations of rock-unit **SS1** and boundary **B2** are predicted to be 17°, not 12° as depicted in the goal diagram. The debugger regresses the desired value of 12° through the dependencies in Figure 39 (the predicted values are in boldface, the desired ones in italics), back to the assumption **Parameter-of(Tilt1, Theta, Theta1)**. The regression indicates that the desired value of **Tilt1**'s **Theta** parameter is 7°, since the regressed value of **SS1.orientation@Tilt1.start + Theta1** is 7° and the predicted value of **SS1.orientation@Tilt1.start** is 0°. Since the desired value of 7° is consistent with the tilt event's parameter constraints, the debugger creates an occurrence repair indicating that the new tilt parameter of **Tilt1** equals 7°.

For the other parameter-binding assumption in Figure 39, regression indicates that the constraint on **Tilt2**'s **Theta** parameter value is:

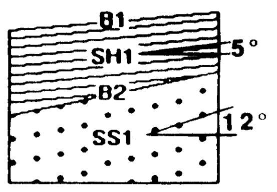
Theta=0 and Theta≠5 and Theta≠0,

where the last conjunct is a parameter constraint of the tilt event. The debugger evaluates this constraint and determines that it is inconsistent, indicating that there is no way of altering **Theta2** to repair the bug.

There are two enhancements to this repair strategy that we have considered, but not yet implemented — 1) suggesting a parameter value that only partly repairs the bug, and 2) trying to further constrain the parameter value using other dependency paths. The first enhancement would be applicable if the parameter-value constraint is inconsistent, as in the above example. In such cases, the debugger would choose a value that lies between the parameter's current value and its desired value. In the above example, for instance, the debugger could create a new tilt parameter whose value is greater than zero but less than five. Although this strategy does not help much in this example, it may be the right thing to do in certain situations.

A second potential enhancement is to further constrain the parameter-value constraint using the regressed values along dependency paths supporting other top-level goals. The debugger would trace forward in the dependencies to find top-level goals supported by the parameter-binding assumption. It would then regress their desired values back along the dependency paths and add any regressed constraint that is not inconsistent with the current parameter-value constraint. In replacing the **Elevel** parameter of an erosion event, for example, we would like the debugger to use the desired values of all goals that depend on this parameter's value to produce a tighter bound on the desired level of erosion.

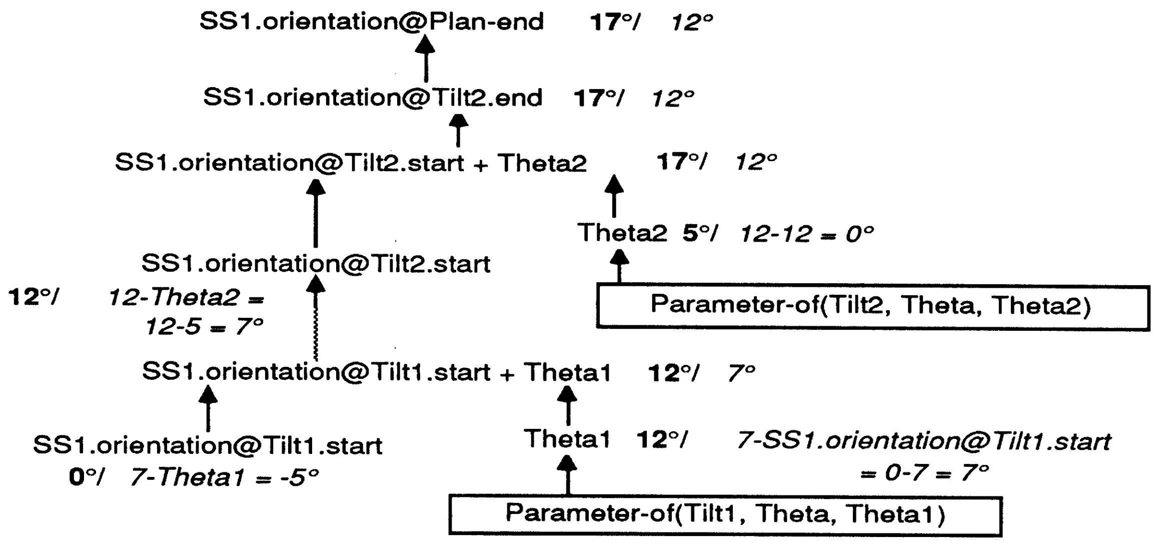
Goal Diagram



Current, Buggy Hypothesis

1. Deposition1(SS1, Sandstone)
2. Tilt1(Theta1=12°)
3. Deposition2(SH1, Shale)
4. Tilt2(Theta2=5°)

Partial Dependency Structure (**Predicted**/*Desired* Values)


SS1.orientation@Plan-end 17°/ 12°
SS1.orientation@Tilt2.end 17°/ 12°
SS1.orientation@Tilt2.start + Theta2 17°/ 12°
Theta2 5°/ 12-12 = 0°
SS1.orientation@Tilt2.start
12°/ 12-Theta2 = 12-5 = 7°
Parameter-of(Tilt2, Theta, Theta2)
SS1.orientation@Tilt1.start + Theta1 12°/ 7°
SS1.orientation@Tilt1.start
0°/ 7-Theta1 = -5°
Theta1 12°/ 7-SS1.orientation@Tilt1.start = 0-7 = 7°
Parameter-of(Tilt1, Theta, Theta1)

Bugs: Orientations of **SS1** and **B2** are 17°, not 12° as measured in the goal diagram.

Figure 39. Replacing a Real-Valued Parameter

4.2.2.4 Replacing Temporal-Ordering Assumptions

The debugger handles temporal-ordering assumptions of the form **Event1.end < Event2.start** by reordering the two events. All three stages of GTD can make such assumptions — the generator and debugger when constructing hypotheses, and the tester when linearizing hypotheses before simulating them.

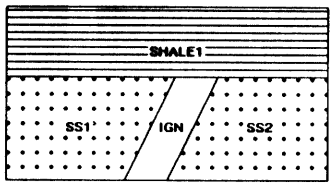
GORDIUS makes two other types of temporal-ordering assumptions that are considered to be too basic to be changed by the debugger (this is indicated by justifying the assumptions with a high degree of belief). One type is the commonsense assumption that the start of an interval always precedes its end (i.e., **I.start < I.end**). The other type is part of the problem statement that, to be considered a solution, an event must fall between **Plan-start** and **Plan-end**.

Temporal-ordering assumptions play two important roles in our causal dependency structures — supporting statements of the form **Persistence(object.attribute, t1, t2)** and **Exists-at(object, t1)**. For **Persistence** statements, the belief that an attribute persists in value from time **t1** to **t2** depends in part on the belief that **t1** in fact precedes **t2** (i.e., **t1 < t2**). For **Exists-at** statements, the belief that an **object** exists at time **t1** is supported by the beliefs that the object was created no later than **t1**, and that **t1** precedes the time, if any, that the object was destroyed (i.e., **object.start < t1** and **t1 < object.end**).

The repair strategy first determines whether the **Persistence** or **Exists-at** statement has alternative supports, in which case removing the temporal-ordering assumption will not help in repairing the bug. This is checked by retracting the assumption, propagating the retraction through the dependency structure (via the TMS), and seeing whether the **Persistence** or **Exists-at** statement still holds. For example, if the generator has asserted that **Deposition2.start < Intrusion1.start**, the statement **Exists-at(SHALE1, Intrusion1.start)** in Figure 40 still holds even after retracting the assumption **Deposition2.end < Intrusion1.start**. Thus, replacing the assumption will not repair the bug.

After retracting the temporal-ordering assumption, the debugger adds new assumptions to reorder events. If the retracted assumption supported an **Exists-at(obj, t1)** statement, a new temporal ordering is added that reverses the previous ordering between time **t1** and the event that created (or destroyed) **obj**. For our causal dependency structures, the correct ordering to be added can be determined by tracing forward in the dependencies from the retracted assumption to the penultimate support of the **Exists-at** statement. The negation of that statement is a temporal ordering that repairs the bug.

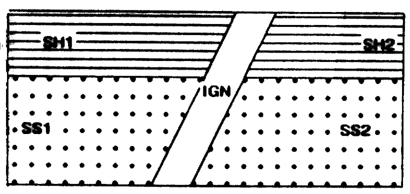
Goal Diagram



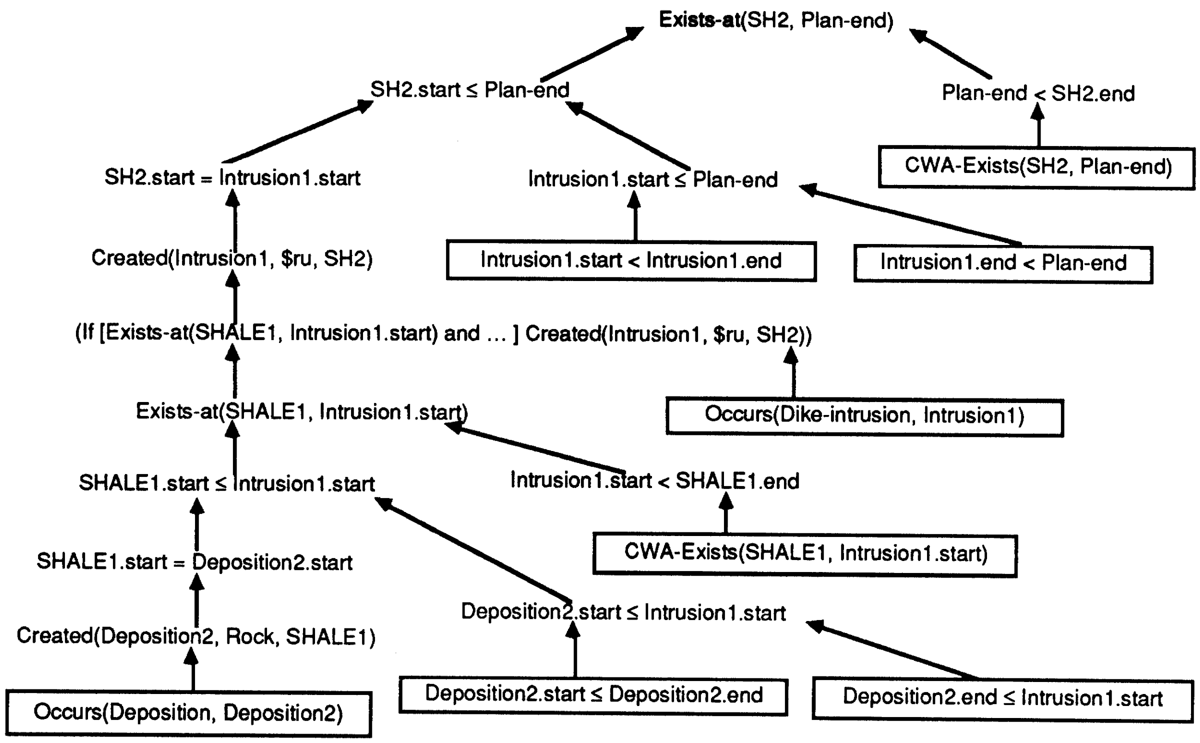
Current, Buggy Hypothesis

1. **Deposition1**(ROCK1, Sandstone)
2. Deposition2(SHALE1, Shale)
3. Intrusion1(IGN, **Mafic-Igneous**)

Simulation Diagram



Partial Dependency Structure


Exists-at(SH2, Plan-end)
SH2.start ≤ Plan-end
Plan-end < SH2.end
CWA-Exists(SH2, Plan-end)
SH2.start = Intrusion1.start
Intrusion1.start ≤ Plan-end
Created(Intrusion1, $ru, SH2)
Intrusion1.start < Intrusion1.end
Intrusion1.end < Plan-end
(If [Exists-at(SHALE1, Intrusion1.start) and ...] Created(Intrusion1, $ru, SH2))
Exists-at(SHALE1, Intrusion1.start)
Occurs(Dike-intrusion, Intrusion1)
SHALE1.start ≤ Intrusion1.start
Intrusion1.start < SHALE1.end
SHALE1.start = Deposition2.start
CWA-Exists(SHALE1, Intrusion1.start)
Deposition2.start ≤ Intrusion1.start
Created(Deposition2, Rock, SHALE1)
Occurs(Deposition, Deposition2)
Deposition2.start ≤ Deposition2.end
Deposition2.end ≤ Intrusion1.start

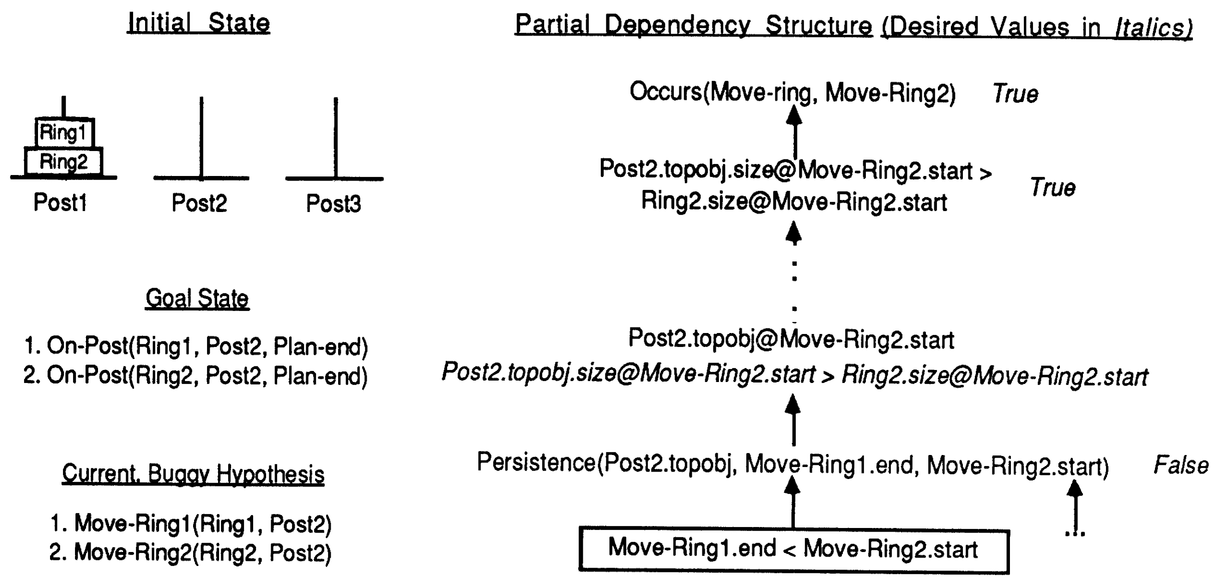
Bug: Extra piece of shale (**SH2**) exists in simulation diagram

Figure 40. Reordering Events to Remove an Extra Piece of Rock

In the example of Figure 40, the diagram matcher detects the bug that an extra piece of shale (**SH2**) appears in the simulation diagram. The dependency structure for this bug explains that the existence of **SH2** depends on the fact that the dike-intrusion created **SH2** by splitting the already existing formation **SHALE1**. The debugger retracts the assumption **Deposition2.end < Intrusion1.start** and replaces it with **Intrusion1.start < Deposition2.start**, the negation of the penultimate statement before reaching **Exists-at(SHALE1, Intrusion1.start)** along the dependency path from the retracted assumption. Adding this assumption repairs the bug because it implies that **SHALE1** will not be split into **SH1** and **SH2**, since it will not exist at the time of the intrusion.

A similar repair strategy is used for temporal-ordering assumptions that support statements of the form **Persistence(obj.attribute, event.end, t1)**, where **event** is the last one to have affected the **attribute** before time **t1**. The repair strategy negates the persistence by reordering so that the end of the persistence interval (**t1**) actually precedes its start (**event.end**).

Unlike the repair strategy for **Exists-at**, however, this one is not applicable in all situations. The strategy is applicable only if the desired value of **obj.attribute@t1** is achieved, or is closer to being achieved, at the start of the persistence interval (i.e., before the **event** occurs). If the strategy is applicable, the bug can be repaired by adding the ordering assumption that **event** occurs after time **t1**.



Initial State
Ring1
Ring2
Post1
Post2
Post3
Goal State
1. On-Post(Ring1, Post2, Plan-end)
2. On-Post(Ring2, Post2, Plan-end)
Current, Buggy Hypothesis
1. Move-Ring1(Ring1, Post2)
2. Move-Ring2(Ring2, Post2)
Partial Dependency Structure (Desired Values in *Italics*)
Occurs(Move-ring, Move-Ring2) *True*
Post2.topobj.size@Move-Ring2.start > Ring2.size@Move-Ring2.start *True*
Post2.topobj@Move-Ring2.start
Post2.topobj.size@Move-Ring2.start > Ring2.size@Move-Ring2.start
Persistence(Post2.topobj, Move-Ring1.end, Move-Ring2.start) *False*
Move-Ring1.end < Move-Ring2.start

Bug: **Move-Ring2** cannot occur because the ring on top of **Post2** (**Ring1**) is smaller than **Ring2**

Figure 41. Reordering Events to Change Persistence of Attribute

For example, in the Tower of Hanoi problem illustrated in Figure 41, the bug in the current hypothesis is that **Move-Ring2** cannot occur because it has an unachieved precondition — the top object on **Post2** must be larger than **Ring2** (see Appendix C). The bug arises because the top object of **Post2** is currently the smaller object **Ring1**. The reordering strategy is applicable here because the desired value of **Post2.topobj** is achieved at the start of **Move-Ring1** — before **Ring1** was moved onto **Post2**, the top object was the base of the post, which is defined to be larger than any ring. The debugger suggests retracting **Move-Ring1.end < Move-Ring2.start** and assuming **Move-Ring2.start ≤ Move-Ring1.end** (i.e., **Move-Ring2** begins before **Move-Ring1** ends).

For simplicity, GORDIUS actually modifies the assumptions suggested by the repair strategies to ensure that events do not overlap. For example, instead of assuming **Move-Ring2.start ≤ Move-Ring1.end**, as above, GORDIUS assumes **Move-Ring2.end < Move-Ring1.start** (i.e., **Move-Ring2** totally precedes **Move-Ring1**). While having one event begin before another one ends is a sufficient repair, given the semantics of our causal models, GORDIUS ensures that no overlap occurs to avoid the possibility of having to reason about simultaneous, interacting events, since this is beyond our current models.

4.2.2.5 Replacing Attribute-Persistence Assumptions

The attribute-persistence assumption — **CWA(object.attribute, t1, t2)** — is a closed-world assumption indicating that no known event changes the value of the **attribute** during the interval from times **t1** to **t2**. Such assumptions are made when determining the value of an attribute at a point in time. GORDIUS finds the interval in the history time-line of the attribute that contains the time point (**t2**) and, if the interval is quiescent, asserts that the value of the attribute is the same as its value at the start of the interval (**t1**), supported by the belief that the attribute persists over the interval.

The repair strategy for an attribute-persistence assumption invalidates the assumption by hypothesizing that some event does affect the attribute during that interval. This can be done by 1) hypothesizing that a new event occurs that affects the attribute, or 2) finding an existing event that has the appropriate effect and reordering it to occur within the persistence interval.

The repair strategy begins by producing the set of occurrence repairs that can achieve the regressed value of **obj.attribute@t2**. Input to the technique for producing occurrence repairs are the temporal reference **obj.attribute@t2**, its desired value, and a list containing each type of event, with the events constrained to end during the persistence interval (i.e., between times **t1** and **t2**).

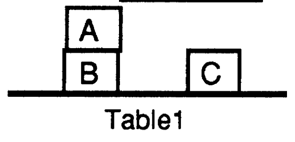
The resulting occurrence repairs represent all the ways that inserting an event can fix the bug. The debugger creates and evaluates new hypotheses for each occurrence repair. In Section 4.1, for instance, GORDIUS fixes the bug that erosion does not occur by determining that uplift and tilt are the two events that can increase the height of the Earth's surface. Both are suggested as repairs and are evaluated to determine their overall goodness.

The repair strategy also checks for existing events that could potentially affect the attribute, but currently do not because 1) they occur outside the persistence interval and/or 2) they have insufficient constraints on their parameter bindings. To find existing events that can fix the bug, the repair strategy relaxes the temporal constraints on the new occurrence repairs and determines which, if any, of the new events unify with existing events. Recall from Section 2.2.2 that two events potentially unify if their types are compatible, their parameter bindings are consistent with one another, and they are currently unordered. Two events necessarily unify if they potentially unify and they create at least some of the same objects.

If a new event and an existing event unify, the debugger creates an occurrence repair to suggest that the existing event is a possible repair. This occurrence repair consists of the existing event plus all parameter binding constraints found for the new event. In addition, if the existing event does not currently fall within the persistence interval, it is reordered using the methods described for the temporal-ordering repair strategy (Section 4.2.2.4).

If a new event potentially, but not necessarily, unifies with an existing event then occurrence repairs are suggested for both the new event and the existing event. This reflects our desire that the debugger should explore all possible options. For example, in Figure 42 the debugger has achieved the goal of clearing **B** by proposing the **Puton1** event to put **A** somewhere. In trying to achieve the goal of having **A** on **C**, the debugger produces an occurrence repair for the event **Puton2(A, C)**. Since this potentially unifies with the existing **Puton1** event, the debugger proposes two repairs — one achieves both goals with the single event of putting **A** on **C**; the other achieves each goal with a separate **Puton** event. For this problem, the former hypothesis is preferred since it is shorter. The latter hypothesis would be preferred, however, if the problem were to stack **A** on **C** on **B**, in which case the **Puton** event used to clear block **B** should be different from the one used to stack **A** on **C**.

Initial State

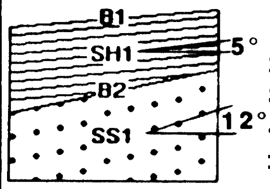


Goal State	Current Hypothesis
1. Clear(B, Plan-end)	1. Puton1(source=A)
2. On(A, C, Plan-end)	

Figure 42. An Occurrence Repair Potentially Unifies with an Existing Event

If a new event and an existing event necessarily unify then both events cannot co-exist in the same hypothesis. In this case, the existing event subsumes the newly proposed event — an occurrence repair is created for the existing event, and the new event is eliminated from further consideration. If, on the other hand, a new event necessarily unifies with more than one existing event, the new event is inconsistent with the current hypothesis, so no occurrence repairs are suggested.

Environment



Goals	Current Hypothesis
1. Composition(SS1, Sandstone, Plan-end)	1. Deposition1(SS1, Sandstone)
2. Composition(SH1, Shale, Plan-end)	2. Deposition2(SH1, Shale)
3. Abuts(B2, SH1, Plan-end)	
4. Abuts(B2, SS1, Plan-end)	
: :	

Figure 43. An Occurrence Repair Necessarily Unifies with an Existing Event

The example in Figure 43 shows how the concept of necessarily unifying events helps in constructing a correct solution.[5] At this point in the example, the debugger has proposed that the two **Composition** goals can be achieved by deposition of **Sandstone**, creating rock-unit **SS1**, and by deposition of **Shale**, creating **SH1**. In trying to achieve the first **Abuts** goal, the debugger produces an occurrence repair that indicates that some deposition event

[5] This example, which is run without using the generator, is described more fully in Section 5.1.2.

(**Deposition3**) created a depositional boundary (**B2**) between the bottom of **SH1** and whatever was on the Earth's surface at the time of the deposition. Since this event necessarily unifies with **Deposition2(SH1, Shale)** (both events purport to create the same rock-unit **SH2**), the occurrence repair for **Deposition3** is replaced with one for **Deposition2** that includes the parameter-binding constraint that **B2** is the lower depositional boundary of **Deposition2**.

4.2.2.6 Replacing Object-Existence Assumptions

The object-existence assumption — **CWA-exists(object, t1)** — is a closed-world assumption indicating that the **object** continues to exist at time **t1** since no event is known to have destroyed it. The assumption is made when evaluating an **Exists-at** statement to determine whether the **object** exists at time **t1**. In our dependency structures, a **CWA-exists** assumption always directly supports an ordering statement of the form **t1 < object.end**.

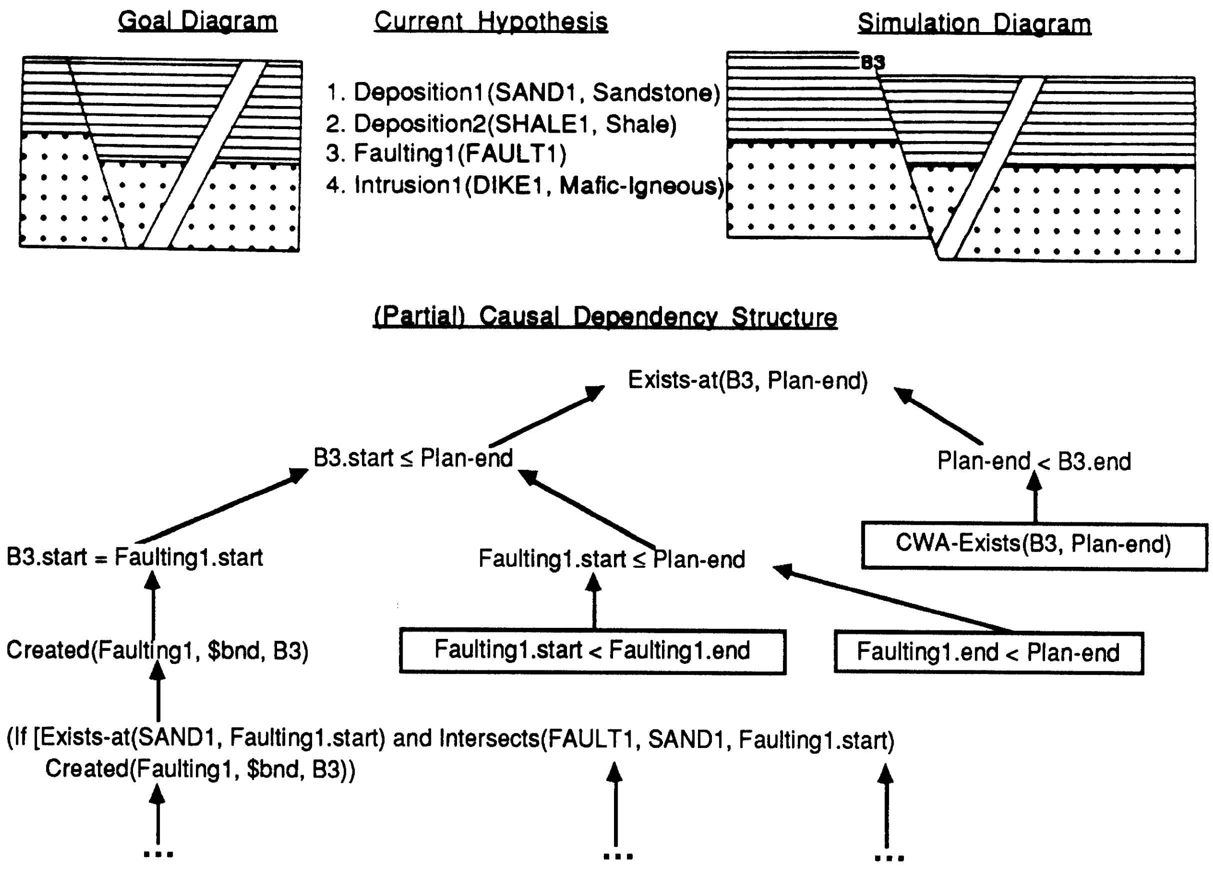
The object-existence assumption, which is similar in meaning to the attribute-persistence assumption, has a similar repair strategy as well. The repair strategy uses the technique for producing occurrence repairs (Section 4.2.2.1) to find the set of events that can destroy the **object**. The events are assumed to occur before time **t1** and after the **object** was created (since objects cannot be destroyed before they are created). If any existing events unify with the new events, they are reordered to occur before **t1** and are suggested as possible repairs.

Finding events that can destroy an object is slightly different from finding events that can affect an object's attribute. One difference is that instead of looking for **Change** statements that affect the attribute, the system looks for statements of the form **Destroyed(object, event.end)** (for efficiency in retrieval, such effects are indexed under the "destroyed" attribute of the event).

The other difference is that there is no magnitude for a **Destroyed** effect — the effect is applicable if 1) the object in the **Destroyed** statement matches the object to be destroyed, and 2) all of the conditions associated with the **Destroyed** effect are potentially achievable. For example, if **B3** (a piece of the fault boundary in Figure 44) is the object to be destroyed, then the effect

```
        (For-all ($gf : geologic-feature)
                (If Is-type($gf, rock-unit) Destroyed($gf, event.end)))
```
is not applicable because the condition **Is-type(B3, rock-unit)** is necessarily false.

The repair strategy begins by inputting to the technique for producing occurrence repairs the object to be destroyed and a list consisting of each type of event, with each event constrained to occur between times **t1** and **object.start**. For each occurrence repair, the debugger creates an hypothesis that adds the new event. It also tries to unify the new event with existing events, using the method described in Section 4.2.2.5 — setting up new occurrence repairs for potentially unifiable events, removing new events that are necessarily unifiable with existing events, and reordering unifiable existing events so that they occur after the object is created and before time **t1**.


Goal Diagram
Current Hypothesis
1. Deposition1(SAND1, Sandstone)
2. Deposition2(SHALE1, Shale)
3. Faulting1(FAULT1)
4. Intrusion1(DIKE1, Mafic-Igneous)
Simulation Diagram
B3
(Partial) Causal Dependency Structure
Exists-at(B3, Plan-end)
B3.start ≤ Plan-end
Plan-end < B3.end
CWA-Exists(B3, Plan-end)
B3.start = Faulting1.start
Faulting1.start ≤ Plan-end
Created(Faulting1, $bnd, B3)
Faulting1.start < Faulting1.end
Faulting1.end < Plan-end
(If [Exists-at(SAND1, Faulting1.start) and Intersects(FAULT1, SAND1, Faulting1.start)
Created(Faulting1, $bnd, B3))
...
...
...

<u>Bug</u>: Extra piece of fault boundary (**B3**) exists in simulation diagram.

Figure 44. Destroying an Object to Remove an Extra Boundary Piece

Figure 44 illustrates a case where this repair strategy is applicable. The diagram matcher finds correspondences for all the pieces in the simulation and goal diagrams except for one extra piece, boundary **B3**. Tracing back through the dependencies, the debugger finds that the assumption **CWA-exists(B3, Plan-end)** underlies the bug. To replace this assumption, the debugger looks for occurrence repairs that can destroy **B3**. The only relevant effect found in our geologic models is:

```
    (For-all ($gf : geologic-feature)
        (If [ Exists-at($gf, Erosion.start) and ($gf ≠ Surface) and
                Elevel ≤ $gf.bottom.height@Erosion.start]
            Destroyed($gf, Erosion.end))),
```

which indicates that erosion destroys any existing geologic feature that is totally above the level of erosion, except for the surface of the Earth.

Since this effect is applicable for destroying boundary **B3**, the debugger suggests inserting a new erosion event (**Erosion1**) constrained to occur before **Plan-end** and after the start of **Faulting1** (the time when **B3** was created). In addition, the debugger assumes that the currently unachieved condition of the **Destroyed** effect holds:

Elevel ≤ B3.bottom.height@Erosion1.start.

This condition provides the diagrammatic simulator with the necessary constraints on how much erosion it needs to perform.

4.2.2.7 Replacing Known-Objects Assumptions

The known-objects assumption — **CWA-object(type, O_1,..., O_n)** — is a closed-world assumption that the only objects of **type** that exist are O_1 – O_n. This assumption is used in evaluating quantified statements. GORDIUS handles a universally quantified statement of the form **(For-all (x : type) P(x))** by expanding it into:

P(O_1) and ... and P(O_n) and CWA-object(type, O_1, ..., O_n).

Similarly, the expansion for **(Exists (x : type) P(x))** is:

P(O_1) or ... or P(O_n) or (not CWA-object(type, O_1,..., O_n)).

The bug repair strategy for a **CWA-object** assumption is to create a new object of the appropriate type. The repair strategy is applicable in two cases: 1) if the object to be created is a type of event, the debugger adds a new event occurrence of that type, and 2) if there is a type of event that can create an object of the appropriate type, the debugger inserts an event of that type together with any parameter bindings needed to create the object. This repair strategy is applicable for creating rock-units and boundaries, for instance, but not for time points.

The debugger also assumes that the newly created the object or event has the necessary properties to satisfy the quantified statement. To achieve the statement:

(Exists ($ru : Rock-Unit) Is-Sedimentary($ru)),

for instance, the debugger would try to add an event that creates a new rock-unit **R1** and assume that **Is-Sedimentary(R1)** is true. Thus, inserting a deposition event would work, but inserting an intrusion event would not since it is inconsistent for an intruded formation to be sedimentary.

As a concrete example of handling quantified goals, suppose given the problem in Figure 45a, we add the goal that some sandstone formation was partially eroded (perhaps we've found chunks of sandstone in a nearby river bed). This goal can be represented by:

```
(Exists ($ru : Rock-Unit)
   ($ru.material@Plan-end = Sandstone) and
   (Exists ($e : Erosion) ($e.Elevel < $ru.top.height@$e.start))),
```

where the inequality indicates that **$e.Elevel**, the level of erosion, is enough to affect the rock-unit **$ru**.

To interpret the diagram, the generator produces the hypothesis of Figure 45b. Simulating this hypothesis, the tester detects that the quantified goal is not achieved since 1) the level of erosion for the existing erosion event (**Elevel1**) is insufficient to affect the existing sandstone rock-unit **SS1**, and 2) the other existing rock-unit (**SH1**) is made of shale, not sandstone. Figure 45c shows relevant portions of the dependency structure for this bug (values predicted by the tester are in boldface).

The known-objects repair strategy is applicable for both **CWA-object** assumptions in the dependency structure. The debugger suggests replacing **CWA-object(Erosion, Erosion1)** with the assumption that another erosion event occurs (**Erosion2**). The debugger assumes that:

Erosion2.Elevel < SS1.top.height@Erosion2.start,

which ensures that the new erosion event affects **SS1**. Another suggested repair replaces the assumption **CWA-object(Sedimentary, SS1, SH1)** by hypothesizing that a new deposition event created another sandstone formation that was subsequently eroded by **Erosion1**.

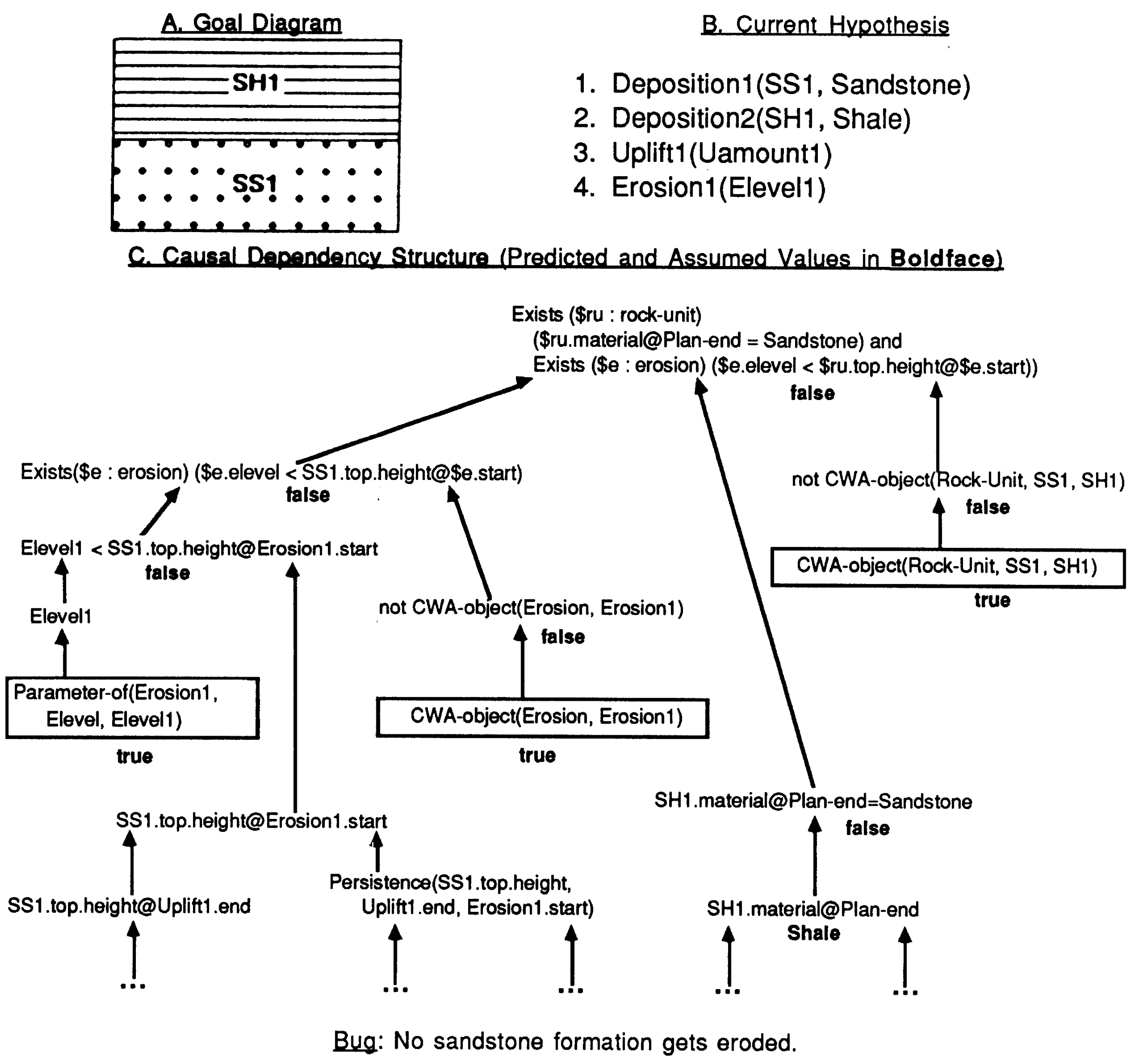

A. Goal Diagram
SH1
SS1
B. Current Hypothesis
1. Deposition1(SS1, Sandstone)
2. Deposition2(SH1, Shale)
3. Uplift1(Uamount1)
4. Erosion1(Elevel1)
C. Causal Dependency Structure (Predicted and Assumed Values in Boldface)
Exists ($ru : rock-unit) ($ru.material@Plan-end = Sandstone) and Exists ($e : erosion) ($e.elevel < $ru.top.height@$e.start))
false
Exists($e : erosion) ($e.elevel < SS1.top.height@$e.start)
false
not CWA-object(Rock-Unit, SS1, SH1)
false
CWA-object(Rock-Unit, SS1, SH1)
true
Elevel1 < SS1.top.height@Erosion1.start
false
Elevel1
not CWA-object(Erosion, Erosion1)
false
Parameter-of(Erosion1, Elevel, Elevel1)
true
CWA-object(Erosion, Erosion1)
true
SH1.material@Plan-end=Sandstone
false
SS1.top.height@Erosion1.start
SS1.top.height@Uplift1.end
Persistence(SS1.top.height, Uplift1.end, Erosion1.start)
SH1.material@Plan-end
Shale
Bug: No sandstone formation gets eroded.

Figure 45. Repairing a Buggy Quantified Goal Statement

The debugger, of course, suggests several other repairs based on the strategies discussed in previous sections. For example, it suggests making **Erosion1** affect **SS1** by replacing the **Elevel1** parameter with a value that is less than the height of the top of **SS1** (this, however, introduces the bug that **SH1** gets eroded away). Similar repairs suggest raising the height of **SS1** above **Elevel1** by changing the amount of uplift, increasing the height of **SS1** by adding another uplift event, etc. The debugger even suggests changing the material parameter of **Deposition2** from shale to sandstone, so when **Erosion1** affects **SH1** it will be eroding a sandstone formation.

The large number of possible bug repairs highlights the importance of controlling search by evaluating the goodness of hypotheses, the subject of the next section.

4.2.3 Control of Search and Evaluating Hypotheses

Our debugging methodology typically suggests many repairs for each bug. The system uses best-first search and a heuristic goodness metric to control the search for a solution.

The primary component of the goodness metric is the number of bugs remaining in the hypothesis, where a bug can be an unachieved top-level goal, an event occurrence that has unachieved preconditions, an unachieved change statement condition, etc. The number of remaining bugs is often a good estimate of the amount of debugging effort needed to find a solution since our task is to find an executable sequence of events that can achieve all the goals of the problem.

The secondary component of the goodness metric is the cost of the hypothesis. This component is only used to differentiate two hypotheses that have the same number of remaining bugs. Currently, the cost of an hypothesis is simply the number of hypothesized events. A more robust approach might calculate the cost of an event as a function of its type and parameter bindings. For example, by making the cost of **Uplift** proportional to the amount of uplift, the system would prefer hypotheses that did as little uplift as possible.

Our experiments indicate that this goodness metric performs very well for the types of problems encountered in the geologic and blocks-world domains. We attribute this mainly to the fact that these domains contain few garden paths, so hill climbing toward the goal state is usually the right thing to do. This conjecture seems to be confirmed by the system's relatively poor performance in solving the Tower of Hanoi problem, a domain in which the correct strategy often involves undoing previously achieved goals.

The goodness metric used by the debugger is essentially the same as the one used by the generator. The major difference is in the method used to calculate the number of remaining bugs. While the generator presumes independence among scenarios, the debugger evaluates each proposed hypothesis to determine its global effects, effects that might include introducing new bugs or serendipitously fixing other existing bugs.

The debugger's evaluation heuristic is based on the causal simulator described in Section 3.2.2, using the same domain models and some of the same methods to evaluate statements. It differs in that it can handle partially ordered hypotheses, but it is not complete, occasionally failing to predict when changes happen to an attribute. We are currently working on ways of minimizing or eliminating that situation while maintaining some degree of efficiency in evaluating hypotheses.

To facilitate the system's search, the evaluation heuristic indicates which orderings of the hypothesis are relevant in determining the number of remaining bugs. For example, suppose our problem is to have blocks **A** on **B**, **B** on **C**, and **D** on **E**, and the current hypothesis consists of three unordered events: **Puton1(A, B)**, **Puton2(B, C)**, and **Puton3(D, E)**. The evaluation heuristic reports that ordering **Puton2** before **Puton1** produces fewer bugs than the other way around, but that the ordering of **Puton3** with respect to the other two events does not affect achievement of the goals.

To facilitate the incremental nature of our evaluation heuristic, the causal dependency structure is implemented on top of a standard monotonic, justification-based TMS [McAllester]. Retracting assumptions causes the inferences supported by the assumptions to be retracted as well. Adding assumptions causes new inferences to be propagated using the local rules of the causal simulator described in Section 3.2.2.

One problem with this scheme is that propagating the effects of events is expensive, especially for non-linear hypotheses. To alleviate this problem, our method does not propagate all effects, only those that are needed to predict changes in the number of remaining bugs. For example, while erosion affects the thickness of rock-units, this effect can be ignored in solving geologic interpretation problems.

Which effects to propagate are determined by analyzing each closed-world assumptions (**CWA**, **CWA-exists**, and **CWA-object**) in the dependency structure to see whether it still holds. For example, GORDIUS checks whether the modifications to the hypothesis affect the attribute of any **CWA** assumption during its persistence interval.

The system retracts each invalidated closed-world assumption and recalculates the dependencies for the statements that were directly supported by that assumption.

For retracted **CWA** assumptions, GORDIUS uses a polynomial time algorithm to recalculate the dependencies of temporal references. The algorithm uses the technique for producing occurrence repairs (Section 4.2.2.1) to determine all possible supports for the temporal reference. It then uses the context switching mechanism of our TMS to predict values of the temporal references under each of the total orderings consistent with the hypothesis. The algorithm essentially pushes the exponential work of evaluating hypotheses down to the TMS. This is a reasonable design decision because current TMS implementations are able to switch contexts very rapidly, with little overhead [McAllester].

Once all the necessary dependencies have been recalculated, the remaining bugs are determined by propagating the changes to the dependency structure using the causal simulation rules. For example, if the value of **A.top@t1** were recalculated, this would propagate to affect the value of the goal **Clear(A, t1)** since **Clear** is defined as **A.top@t1 = {}**. The number of remaining bugs is then calculated simply by counting the number of inconsistent TMS nodes in our causal dependency structure.

One additional characteristic of the debugger's search algorithm is that it must check for cycles in the search tree. Cycles might appear, for instance, if the debugger inserted an event and then deleted it. To prevent the debugger from looping in such situations, the debugger examines the ancestor nodes of each hypothesis to determine if it is equivalent to one that had been suggested previously. Two hypotheses are considered equivalent if the events in each hypothesis unify with at least one event in the other hypothesis.

4.3 A Theory of Debugging

The assumption-oriented debugger described in this chapter provides the foundation for a general theory of debugging plans and interpretations. The basic idea is that all bugs ultimately arise from faulty assumptions made during the construction and testing of hypotheses. The debugger repairs inconsistencies between the predicted and desired states of the world by identifying and replacing faulty assumptions that underlie the inconsistencies.

The theory conjectures that there is only a relatively small number of different types of assumptions that can underlie bugs. It further conjectures that the many ways that bugs can arise through different combinations of assumptions can be handled by using domain-independent methods for tracing through causal dependency structures, regressing values of statements, and determining how to replace assumptions. These three techniques — tracing dependencies, regressing values, and using repair strategies that employ causal reasoning — focus the debugger on which assumptions to replace and how to replace them.

The efficacy of the GTD paradigm rests, to a large extent, on having a robust debugger. The rest of this section analyzes the robustness of our theory of debugging in terms of 1) its degree of completeness, 2) situations in which the debugging algorithm may not terminate, and 3) the extent to which the implemented repair strategies cover the range of assumptions that can possibly underlie a bug.

The conclusions reached for the different aspects of robustness are:

Completeness: for the causal representation language used and the assumptions made explicit in our causal models, the dependency tracing technique is complete, as are the repair strategies, under the presumption that bugs can be fixed by replacing one assumption at a time. Since the regression technique, however, is not in general complete, in theory the debugger might not find a solution when one exists. For the problems we examined to date, however, it has in fact proven to be pragmatically complete.

Termination: the implemented goodness metric and evaluation heuristic need to be improved slightly to ensure that GORDIUS will always halt if a problem has a solution. Since planning and interpretation are undecidable, in general, the debugger is not guaranteed to halt if the problem has no solution.

Coverage: by analyzing three models upon which GORDIUS is built — a model of causality, a model of the problem-solving task, and a model of hypothesis construction — we conclude that our set of repair strategies is sufficient to handle many of the common causes of bugs. In addition, the debugger can easily be extended to handle assumptions currently only implicit in our system. New simulation rules, regression rules, and repair strategies can be added without changing the existing rules, repair strategies, or the debugging algorithm itself.

4.3.1 Completeness

By "completeness" we mean "can the system solve all problems that have a solution and are describable in the representation language used?" The aim of this section is not to offer formal proofs, but rather to characterize the degree of completeness for the dependency tracing, regression, and bug repair techniques.

For a given causal explanation, the dependency tracing technique is complete in that it will find all assumptions underlying a bug that could be at fault. This is because all dependency paths are examined except for those that the pruning methods show cannot possibly be changed to repair the bug.

The caveat, of course, is that the method is only as complete as the causal explanations examined. If an assumption does not appear explicitly in the dependency structure, the debugger has no means of locating it. Similarly, if the explanation does not match the true causality in the domain, faulty assumptions may be overlooked. Questions of how well our explanations actually model reality is the subject of Section 4.3.3.

Although all the causal models used by GORDIUS are deterministic, the debugging theory is applicable to non-deterministic models, as well, as long as the causal explanations constructed include all possible underlying assumptions. For example, in a probablisitic model with rules of the form: "if **X1** then **Y** happens with probability **p1**" and "if **X2** then **Y** happens with probability **p2**," the causal explanations must indicate that **Y** depends on both **X1** and **X2**. Although this might greatly increase the number of assumptions that must be examined, the debugger can perhaps use the probabilities to heuristically order the assumptions according to their *a priori* likelihood of causing the bug.

A major source of incompleteness is the technique that regresses values back through the dependencies. The problem is that most of the repair strategies depend on the regression to constrain the choice of parameter values. If the constraints produced by the regression are under-determined, the debugger must choose and test different parameter values until one is found that solves the problem.

This technique is incomplete when only a finite number of values out of an infinite set (e.g., the reals) can solve the problem, since in general it will take infinite time to test each choice before hitting on a correct solution. While in many cases hill-climbing through the space of parameter values can help to focus in on the correct choice, in the worst case the complete parameter space will have to be examined. 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.

There are several additional difficulties inherent in regressing values. One is that the regressed constraints can become very complex, making it difficult to reason with them. Although this can be alleviated somewhat using better algebraic simplifiers, simplification in general is an unsolvable problem. Another difficulty is that not all dependency links can be symbolically inverted. Although techniques exist to solve some of the problems that GORDIUS cannot currently handle (e.g., expressions of the form **Y = X * X** can be solved by [MACSYMA] or [Sacks]), there are many expressions that cannot be inverted, for instance, those involving higher order polynomials. Even more problematic is that the domain models may contain user-defined functions whose exact definition is not provided. In such cases, the regression gives the debugger almost no information on how to change the function's arguments to achieve its desired value.

Turning to the completeness of the repair strategies, although we do not offer proofs, it should be clear from the descriptions given in this chapter that our repair strategies are complete in determining appropriate ways to replace an assumption, given sufficient regression constraints. For example, a parameter-binding assumption is replaced with one whose parameter value is sufficient to repair the bug, as determined by the regression; an event-occurrence assumption is handled by deleting (or replacing) the event; etc.

One strategy whose completeness we did analyze more formally is the attribute-persistence repair strategy (Section 4.2.2.5). Formally, GORDIUS' model of persistence is:

```
∀(attr, obj, t1, t2)
    {Persistence(obj.attr, t1, t2) ≡
        (t1 < t2) and
        not ∃(type, mag, event) [Change(type, obj.attr, mag, event) and
                                 (t1 < event.end) and (event.end ≤ t2)]},
```

where the negated existential clause states that no event changes the object's attribute between the start and end of the persistence interval.

In the dependency structures produced by the tester, the negated existential clause is represented by an attribute-persistence (**CWA**) assumption. Negating a **CWA** assumption is thus equivalent to making the existential clause true, that is, having some event change **obj.attr** during the persistence interval. By examining the existential clause, we see that this can be accomplished by 1) inserting a new event that has the appropriate **Change** statement, or 2) using an existing event and constraining it to occur within the persistence interval. Both these strategies are tried by the attribute-persistence repair strategy.

The repair strategies are not complete in one important respect — although the debugger *can* repair bugs that depend on multiple faulty assumptions, as long as replacing each assumption separately moves the hypothesis closer to a solution, in general the debugger fails in situations where changing any one of the assumptions separately has no discernible effect on repairing the bug. For example, the debugger cannot handle situations where two events must be added, neither of which has a positive effect by itself (e.g., needing to use two hands to pick up a box). The debugger also cannot handle cases 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 possible way for handling such problems is to use techniques developed to handle multiple points of failure in diagnosis. The technique described in [Davis] efficiently controls the combinatorial search inherent in examining combinations of assumptions by layering the search — examining single assumptions first, and examining combinations only when no solution is found. Although this may be a viable technique for our debugger as well, it is not currently incorporated in GORDIUS because 1) it is difficult to develop general repair strategies that can handle combinations of assumptions, and 2) this is a fairly rare problem (it has not arisen in our domains), and thus has limited impact on the robustness of the debugger.

4.3.2 Termination

It is useful to consider separately whether the debugger terminates when a problem does and does not have a solution. We will see that in either case termination is not guaranteed for GORDIUS. Although non-termination is intrinsic in the case where no solution exists, termination can be guaranteed when a solution exists with simple changes to GORDIUS' current search metric and evaluation heuristic.

Planning and interpretation tasks have been shown to be undecidable for representation languages, such as ours, that model effects that depend on the current state [Chapman]. Thus, for cases where no solution exists, it is provably true that no debugger will always terminate. On the other hand, our debugger may terminate in certain situations — precisely those in which no more headway can be made towards a solution, that is, where no modification can be suggested that even partly achieves any of the problem's goals. A simple illustration of this is where the desired value of some temporal reference is not achieved and GORDIUS knows of no event that affects the attribute of the temporal reference.

The debugger may also terminate without a solution in situations where a bug cannot be repaired by replacing one assumption at a time (see Section 4.3.1). Since these situations are rare, however, termination without a solution usually indicates that no solution exists.

In our current implementation, there are two potential ways in which the debugger may not terminate when a solution in fact exists. Both can be remedied with simple implementation changes, which have not yet been made due to time constraints. One problem is that our current evaluation heuristic does not accurately estimate the number of remaining unachieved goals. In particular, it sometimes fails to detect when a modification serendipitously repairs additional bugs, causing the debugger to overlook potential solutions. The solution here is to implement a more complete evaluation heuristic that avoids such false negatives.

The second potential cause of non-termination is that our current goodness metric is not admissible. This stems from using the number of events only as a tie-breaker in cases where two hypotheses have the same number of unachieved goals. This may cause non-termination in situations where some event helps towards achieving a goal, but no finite number of such events can actually achieve the goal — something that is theoretically possible, but practically unlikely.

For example, it would take an infinite number of events to move an object a finite distance if each event moves it by only an infinitesimal amount. In such cases, the debugger might add such events endlessly, since it would consider that progress was continually being made towards a solution. One way to make the metric admissible is to combine both components into a single measure, so when one hypothesis gets too long another is pursued, even if it has more unachieved goals.

One other potential cause of non-termination is currently handled correctly by GORDIUS. The problem is that the debugger might loop endlessly by repeatedly adding an event to achieve one goal and then deleting it to achieve a different goal. As described in Section 4.2.3, GORDIUS avoids such cycles in the search space by not pursuing an hypothesis if it is equivalent to one already proposed higher up in the search tree.

4.3.3 Coverage and Extensibility

The previous two sections show that the debugger is quite robust in handling faulty assumptions that appear explicitly in our causal dependency structures. The issues discussed in this section are 1) how well the current set of assumptions covers the range of interpretation and planning problems, and 2) how easily the debugger can be extended to handle assumptions now made only implicitly. Both our experience and analysis show that the six types of assumptions currently handled cover a fairly large range of problems. In addition, it is relatively easy to extend the system to handle many new types of assumptions.

In practice, our experiments indicate that the debugger can handle a wide range of bugs in several different domains, arising from many different combinations of assumptions (Chapter 5). More formally, the coverage provided by our current debugger can be characterized by examining the premises underlying three different models upon which GORDIUS is built — a model of causality, a model of hypothesis construction, and a model of the problem-solving task.

Each model provides a representational framework that helps define the scope and expressive power of the problem solver. For example, our model of causality defines the system's understanding of how the world works. The model explicitly represents time and the effects of events, and it presumes that objects and their attributes continue to persist in value unless some event changes them.

Our model of causality indicates that one can predict future states of the world given only an hypothesized set of events and assumptions about the initial state, closed-world assumptions that all objects (including events) are known, and assumptions that the domain models are correct and complete. A bug arises when the predicted state is inconsistent with the assumptions of the goal state.

To provide complete coverage, the debugger must handle all the assumptions involved in predicting effects and detecting bugs — the assumptions made in constructing hypotheses, assumptions about the initial and goal states, closed-world assumptions made by the causal model, and assumptions about the correctness of domain models. We argue below that our debugger has wide coverage, since it currently handles all but the latter assumption and some types of closed-world assumptions.

The hypothesis construction model defines the actions available to the system in forming an hypothesis. The model embodied in GORDIUS indicates that hypotheses are completely specified by the events that occur, the parameter bindings of events, 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.

The problem-solving task model defines what constitute planning and interpretation problems and solutions. A problem is represented by assumptions about the initial and goal states. The task model indicates that these assumptions are fixed and cannot be changed by the debugger, otherwise it would be solving a different problem. The task model defines a solution as a set of events, occurring between times **Plan-start** and **Plan-end**, that achieves all the goals. This definition implies that temporal orderings containing the time points **Plan-start** and **Plan-end** also cannot be changed by the debugger.

The debugger currently handles three types of closed-world assumptions (**CWA**, **CWA-exists**, and **CWA-object**) that are commonly made (and commonly at fault) in predicting states of the world in our domains. Although practical experience has not shown the need for handling others (e.g., the implicit assumption that discrete models are sufficient), it is a fairly simple matter to extend the debugger to handle other closed-world assumptions.

To extend GORDIUS, one needs to add 1) local evaluation rules that both determine the value of a statement based on the values of its arguments and record dependencies based on the evaluation, 2) rules that symbolically regress the value of a statement to determine constraints on the values of its supports, and 3) repair procedures that know how to replace assumptions given dependencies, regressed values, and domain models.

Our experience has shown that in many cases it is relatively easy to add these three types of knowledge to handle new types of assumptions, usually with little need to modify existing simulation rules or debugging repair strategies. For example, we recently added evaluation and regression rules for handling quantified goal statements, and added the known-objects repair strategy to handle the closed-world assumptions (**CWA-object**) that support quantified statements (Section 4.2.2.7). The additions were integrated smoothly into the existing debugger with only almost no change needed to the debugging algorithm itself. The only exception is that the evaluation heuristic was augmented to examine **CWA-object** assumptions when modifications are made to hypotheses. This is necessary because closed-world assumptions are non-monotonic and adding new information might invalidate them.

Not so simple to handle are the implicit assumptions that the domain models are correct and complete. Handling them is somewhat tricky because any bug can be fixed by changing the domain models in an appropriate way. For example, we could debug the example in Section 4.1 simply by eliminating the precondition that erosion must occur below sea-level. 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.

The inability to handle such assumptions has practical consequences because some of our current event models are not in fact complete. In particular, the current faulting and dike-intrusion models incompletely represent the effects of splitting rock-units and boundaries into pieces. This is primarily because we have found it difficult to represent declaratively the complete range of topological and geometrical changes that occur as a result of splitting objects. One consequence is that the debugger does not suggest repairs that would be reasonable had it access to the geologic knowledge currently encoded by the tester's more complete, but procedural, diagrammatic models.

In conclusion, our theory of debugging provides a very robust framework for repairing bugs in plans and interpretations. The debugger is nearly complete, and with simple implementation changes can be guaranteed to terminate when a solution exists. Just as important, it provides good coverage of the common types of faulty assumptions, and is easily extended to handle assumptions not currently made explicitly by GORDIUS. A subject for future work is to examine how well the theory extends to debugging in other tasks, such as design or diagnosis, that have different underlying models of causality, hypothesis construction, and the problem-solving task.

5. Experiments

GORDIUS has been tested in four different domains — our primary domain of geologic interpretation, simple blocks-world planning problems, Tower of Hanoi problems, and diagnosis of manufacturing faults in semiconductor fabrication [Mohammed & Simmons].

5.1 Geologic Interpretation

We have used GORDIUS to solve approximately two dozen geologic interpretation problems. Most were solved correctly by the generator, using the library of 15 scenarios in Appendix A. A representative sample of these problems is presented in Figure 46, which shows the goal diagram and the (linearized) solution generated for each problem.

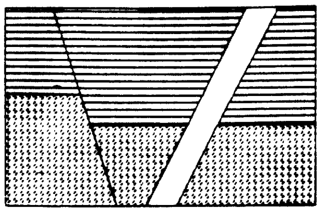
Figure 47 presents statistics of GORDIUS' behavior for the examples of Figure 46. The statistics include 1) the number of goal statements in each problem, 2) the total number of scenarios matched (a measure of the potential size of the search space), 3) the number of hypotheses suggested (equivalent to the number of nodes expanded in the search tree), 4) the number of those hypotheses actually pursued, and 5) the length of the solution path (indicating the number of scenarios needed to account for all the problem's goals). Also presented are timing statistics for the generator and tester.

The "efficiency" statistic (the ratio of hypotheses pursued to solution path length) indicates how well the generator avoids backtracking. This statistic indicates that the generator pursues relatively few false paths, but that it is not perfect. Even so, the generator is very efficient — the time spent searching for a solution is typically only a third of the time spent matching scenarios[1] and is only a fraction of the time spent needed to test the generated hypothesis.

One notable exception to the efficiency of the generator is example B, in which a rather large number of hypotheses are pursued. This happens because one of the first scenarios instantiated ("Intrusion-To-Surface") turns out to be inconsistent. Since the generator

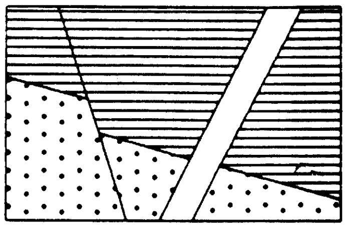
[1] The time needed to match can be reduced significantly using parallel processing.

A.



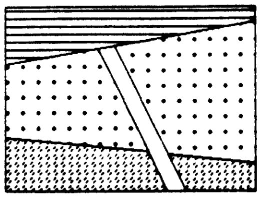
1. Deposition1(Rock1, Shale)
2. Batholithic-Intrusion2(Rock2, Granite)
3. Uplift3(Uamount1)
4. Faulting4(Fault1)
5. Dike-Intrusion5(Ign1, Mafic-igneous)
6. Erosion6(Elevel1)

B.



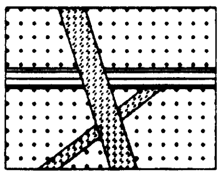
1. Deposition1(Rock1, Sandstone)
2. Deposition2(Rock2, Shale)
3. Uplift3(Uamount1)
4. Tilt4(-5°)
5. Faulting5(Fault1)
6. Dike-Intrusion6(Ign1, Mafic-igneous)
7. Erosion7(Elevel1)

C.



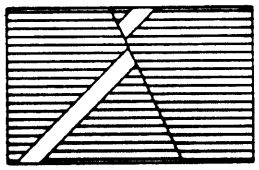
1. Deposition1(Rock1, Sandstone)
2. Tilt2(8°)
3. Batholithic-Intrusion3(Rock2, Granite)
4. Dike-Intrusion4(Ign1, Mafic-igneous)
5. Deposition5(Rock2, Shale)

D.



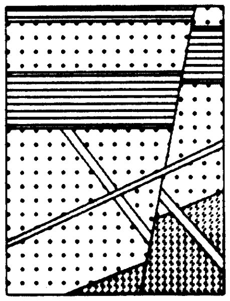
1. Deposition1(Rock1, Sandstone)
2. Dike-Intrusion2(Ign1, Mafic-igneous)
3. Deposition3(Rock2, Shale)
4. Deposition4(Rock3, Sandstone)
5. Dike-Intrusion5(Ign2, Mafic-igneous)

E.



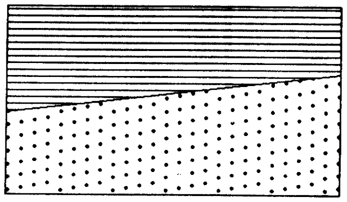
1. Deposition1(Rock1, Shale)
2. Uplift2(Uamount1)
3. Dike-Intrusion3(Ign1, Mafic-igneous)
4. Reverse-Faulting4(Fault1)
5. Erosion5(Elevel1)

F.



1. Deposition1(Rock1, Sandstone)
2. Batholithic-Intrusion2(Rock2, Granite)
3. Dike-Intrusion3(Ign1, Mafic-igneous)
4. Deposition4(Rock3, Shale)
5. Deposition5(Rock4, Sandstone)
6. Deposition6(Rock5, Shale)
7. Faulting7(Fault1)
8. Uplift8(Uamount1)
9. Dike-Intrusion9(Ign2, Mafic-igneous)
10. Erosion10(Elevel1)

G.



1. Deposition1(SS1, Sandstone)
2. Tilt1(9°)
3. Deposition2(SH1, Shale)

Figure 46. Sample of Geologic Interpretation Problems Solved by GORDIUS Generator

Examples	A	B	C	D	E	F	G
1. Goal Statements in Problem	56	56	40	89	50	147	10
2. Scenarios Matched	28	28	29	64	29	108	3
3. Hypotheses Suggested by Generator	30	58	41	72	37	38	3
4. Hypotheses Pursued	16	42	8	15	13	29	2
5. Solution Path Length	13	13	7	13	9	27	2
6. Efficiency (Ratio of 4 to 5)	1.2	3.8	1.1	1.2	1.4	1.1	1.0
7. Effort (Ratio of 3 to 1)	0.5	1.0	1.0	0.8	0.7	0.3	0.3
8. Time Used by Generator (Min:Sec)	:19	:26	:20	:53	:20	2:29	:04
8a. Pattern Matching Time	:14	:14	:14	:47	:14	1:46	:03
8b. Scenario Instantiation Time	:05	:12	:06	:06	:06	:43	:01
9. Time Used by Tester (Min:Sec)	2:54	3:40	1:10	1:27	1:38	5:49	:28
9a. Causal Simulation Time	1:18	1:41	:41	:26	:40	2:43	:17
9b. Diagrammatic Simulation Time	1:36	1:59	:29	1:01	:58	3:06	:11

Figure 47. Statistics for Generating the Geologic Interpretations of Figure 46

Examples	H	I	J	K	L†
1. Total Number of Bugs	1	2	10	1	2
2. Changeable Assumptions	7	15	95	5	18
3. Repairs Suggested by Debugger	3	7	16	5	10
4. Hypotheses Pursued	1	1	6	1	2
5. Solution Path Length	1	1	6	1	2
6. Efficiency (Ratio of 4 to 5)	1.0	1.0	1.0	1.0	1.0
7. Effort (Ratio of 3 to 1)	3.0	3.5	1.6	5.0	5.0
8. Time to Generate Initial Hypothesis (Min:Sec)	:18	:12	N.A.*	:27	:17
9. Time to Test Initial Hypothesis	:34	1:35	N.A.*	2:46	:40
10. Time to Debug Hypothesis	2:17	:42	2:32	1:08	3:20
11. Time to Retest Debugged Hypothesis	2:36	1:29	:29	2:51	N.A.†

* Not applicable — the example was run without using the generator.
† Did not run to completion (see Section 5.1.4); statistics are for part that ran.

Figure 48. Statistics for Debugging the Geologic Problems of Figure 49

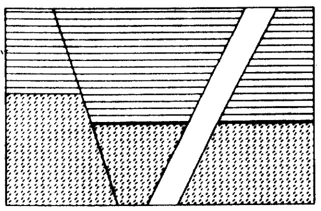
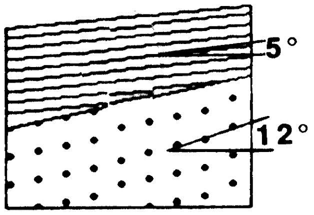
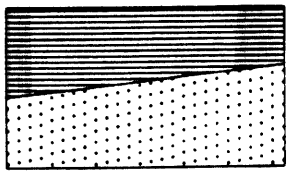
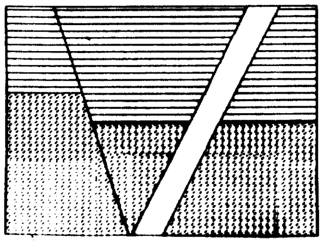
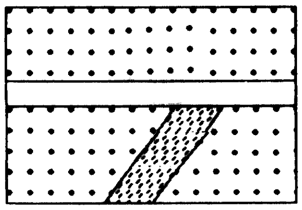
	Goal Diagrams	Initial, Generated Hypotheses	Debugged Solutions
H.		1. Deposition1(ROCK1, Shale) 2. Batholithic-Intrusion1 (ROCK2, Granite) 3. Faulting1(Fault1) 4. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous) 5. Erosion1(EROBOUND, Elevel1)	1. Deposition1(ROCK1, Shale) 2. Batholithic-Intrusion1 (ROCK2, Granite) 3. Uplift1(Uamount1) 4. Faulting1(Fault1) 5. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous) 6. Erosion1(EROBOUND, Elevel1)
I.		1. Deposition1(SS1, Sandstone) 2. Tilt1(12°) 3. Deposition2(SH1, Shale) 4. Tilt2(5°)	1. Deposition1(SS1, Sandstone) 2. Tilt1(7°) 3. Deposition2(SH1, Shale) 4. Tilt2(5°)
J.		(example run without using generator)	1. Deposition1(SS1, Sandstone) 2. Tilt1(9°) 3. Deposition2(SH1, Shale)
K.		1. Deposition1(ROCK1, Shale) 2. Batholithic-Intrusion1 (ROCK2, Granite) 3. Uplift1(Uamount1) 4. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous) 5. Faulting1(Fault1) 6. Erosion1(EROBOUND, Elevel1)	1. Deposition1(ROCK1, Shale) 2. Batholithic-Intrusion1 (ROCK2, Granite) 3. Uplift1(Uamount1) 4. Faulting1(Fault1) 5. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous) 6. Erosion1(EROBOUND, Elevel1)
L.		1. Deposition1(ROCK1, Sandstone) (SS1, SS2, and SS3 are pieces of ROCK1) 2. Dike-Intrusion1(G1, Granite) 3. Dike-Intrusion2 (MI1, Mafic-Igneous)	1. Deposition1(ROCK1, Sandstone) 2. Dike-Intrusion1(G1, Granite) 3. Dike-Intrusion2 (MI1, Mafic-Igneous) 4. Uplift1(Uamount1) 5. Erosion1(Elevel1) 6. Subsidence1(Samount1) 7. Deposition2(ROCK2, Sandstone)

Figure 49. Sample Interpretation Problems Debugged by GORDIUS

currently uses chronological backtracking, GORDIUS explores many intermediate hypotheses before it gets back on the solution path. Once the generator is on the correct solution path, however, it finds a solution in short order. In fact, by changing the order in which the goals are examined, the generator can solve example B by pursuing only 14 hypotheses, rather than 42.

The "effort" statistic (Figure 47, #7) indicates the average number of nodes searched to achieve each goal. For the generator, the "effort" per goal is typically less than one because each scenario can account for multiple goals — our geologic scenarios, for instance, the average six goals per scenario. Achieving multiple goals at each search step is particularly effective when contrasted with the alternative — for geologic interpretation problems, a generator that achieved one goal at a time would suggest an average of four hypotheses per goal (the average number of scenarios that match each goal).

A similar analysis can be made for the behavior of the debugger. Figure 48 presents the debugger's behavior for five interpretation problems (shown in Figure 49), using statistics analogous to those given for the generator. Note that example J is the same as example G in Figure 46 — the difference is that in J, GORDIUS solves the problem with the debugger alone (i.e., the generator's scenario library is empty).

The "efficiency" statistic indicates that the debugger's search is very focused for the problems examined, consistently choosing the correct hypothesis to pursue next. The hill-climbing strategy is successful in these geologic problems because the plausible interpretations are usually those where an event achieves a goal directly. This contrasts with Tower of Hanoi problems and some blocks-world problems, in which the correct solution often involves deliberately undoing previously achieved goals. In those cases the debugger does in fact explore false search paths.

Interestingly, the debugger's search is even more focused than the generator's. This is largely because the debugger, by analyzing interactions between events, finds potential inconsistencies sooner than does the generator. This analysis, however, is expensive — the statistics show that the generator, which uses only simple consistency checks, is about an order of magnitude faster per hypothesis suggested. Overall, the generator's strategy of not reasoning about potential interactions seems to be cost-effective, even given that it sometimes results in searching false paths.

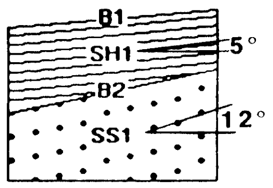
The "effort" statistic reveals another dimension along which the generator solves problems more efficiently than the debugger — it suggests far fewer hypotheses per goal. For example, the debugger proposes over five times as many hypotheses in solving the same problem (examples G in Figure 46 and J in Figure 49). This comparison reflects the efficacy of trying to achieve multiple goals at once. It also argues for using scenarios that encapsulate interactions — many of the repairs suggested by the debugger to achieve one goal interact badly with the rest of the goals. By using encapsulations, GORDIUS can avoid suggesting and evaluating such hypotheses.

The following four sub-sections describe how GORDIUS debugs the problems of I-L in Figure 49 (example H is presented in Section 4.1). The examples were chosen to highlight the capabilities of the GORDIUS debugger, and are intended to show how our theory of debugging can handle seemingly diverse types of bugs and their manifestations.

5.1.1 Double Tilt Example

Figure 50 presents an interpretation problem in which the two sedimentary formations are oriented at different angles. This example was chosen to illustrate the applicability of many of our bug repair strategies. In particular, it demonstrates the utility of the parameter-binding repair strategy (Section 4.2.2.3).

A. The Goal State



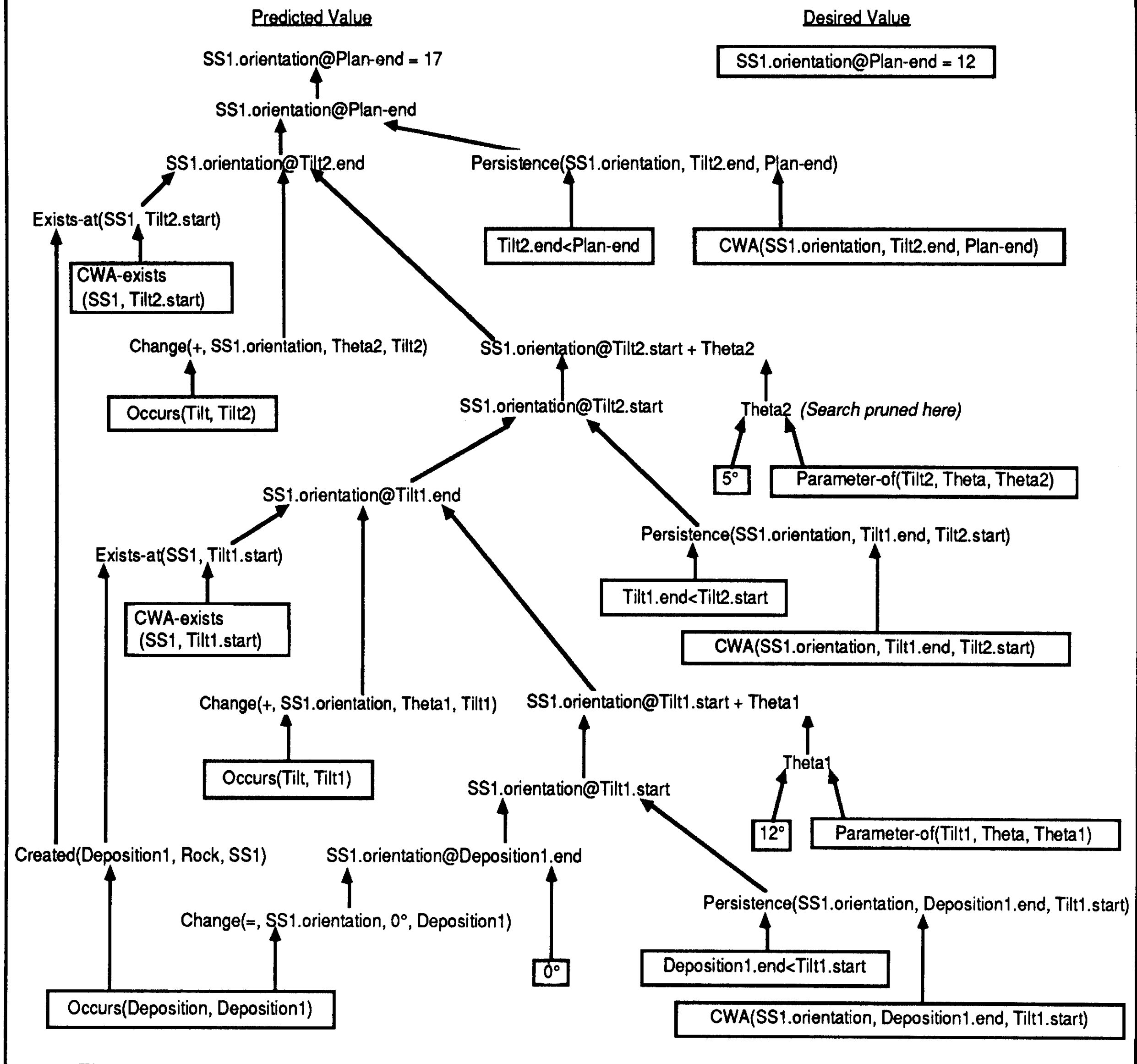
B. Initial Hypothesis Generated

1. Deposition1(SS1, Sandstone)
2. Tilt1(12°)
3. Deposition2(SH1, Shale)
4. Tilt2(5°)

Figure 50. Two Formations with Different Orientations

The generator begins interpreting the diagram by using the "sedimentary-over-rock" scenario (which indicates that an overlying sedimentary formation is younger) to hypothesize that a **Deposition1** event created **SH1** after a **Rock-Creation2** event formed **SS1**. To interpret the orientations of **SS1** and **B2**, the generator uses the "tilted-sedimentary" scenario (which indicates that a sedimentary formation oriented at an angle θ was formed by deposition followed by a tilt of θ). The generator uses this scenario a second time to interpret the orientations of **SH1** and **B1**; it then unifies the hypothesized **Deposition2** event with **Rock-Creation2** since both events purport to create **SH1**. Combining all the constraints (and linearizing the events), the generator produces the initial hypothesis in Figure 50b.

Two bugs are detected in testing the hypothesis — the predicted orientations of both **SS1** and **B2** are 17°, while their desired orientations in the goal diagram are 12°. The debugger first considers the bug that the orientation of **SS1** is not 12°. To locate its underlying assumptions, the debugger traces back through the dependencies in Figure 51 and regresses through them the desired value of 12°.



Predicted Value
Desired Value
SS1.orientation@Plan-end = 17
SS1.orientation@Plan-end = 12
SS1.orientation@Plan-end
SS1.orientation@Tilt2.end
Persistence(SS1.orientation, Tilt2.end, Plan-end)
Exists-at(SS1, Tilt2.start)
Tilt2.end<Plan-end
CWA(SS1.orientation, Tilt2.end, Plan-end)
CWA-exists (SS1, Tilt2.start)
Change(+, SS1.orientation, Theta2, Tilt2)
SS1.orientation@Tilt2.start + Theta2
Occurs(Tilt, Tilt2)
SS1.orientation@Tilt2.start
Theta2 (Search pruned here)
5°
Parameter-of(Tilt2, Theta, Theta2)
SS1.orientation@Tilt1.end
Persistence(SS1.orientation, Tilt1.end, Tilt2.start)
Exists-at(SS1, Tilt1.start)
Tilt1.end<Tilt2.start
CWA-exists (SS1, Tilt1.start)
CWA(SS1.orientation, Tilt1.end, Tilt2.start)
Change(+, SS1.orientation, Theta1, Tilt1)
SS1.orientation@Tilt1.start + Theta1
Occurs(Tilt, Tilt1)
Theta1
SS1.orientation@Tilt1.start
12°
Parameter-of(Tilt1, Theta, Theta1)
Created(Deposition1, Rock, SS1)
SS1.orientation@Deposition1.end
Persistence(SS1.orientation, Deposition1.end, Tilt1.start)
Change(=, SS1.orientation, 0°, Deposition1)
0°
Deposition1.end<Tilt1.start
Occurs(Deposition, Deposition1)
CWA(SS1.orientation, Deposition1.end, Tilt1.start)

Figure 51. Causal Dependency Structure for Bug that the Orientation of **SS1** is not 12°

While regressing values, the debugger uses its pruning methods to determine which nodes need not be pursued. One pruning method is applicable at the node **Theta2** in Figure 51 — the regression indicates that the desired value of **Theta2** is 0°, which is inconsistent a the constraint of the **Tilt** event that its parameter is non-zero (see Appendix B). Thus, dependency tracing is halted at the node **Theta2**, and the assumptions labeled **Parameter-of(Tilt2, Theta, Theta2)** and **5°** are never considered by the debugger.

Of the remaining 15 assumptions located by tracing dependencies back to leaf nodes, four are ignored by the debugger because they are considered to be unchangeable — **SS1.orientation@Plan-end=12°** because it is a goal statement, the values **12°** and **0°** because they are constants, and the ordering **Tilt2.end < Plan-end** because hypothesized events are constrained to occur before **Plan-end**.[2]

For the three **CWA** assumptions, the debugger proposes the same repair of adding a new tilt event of -5° between the start and end of the persistence interval. The repairs, however, are rated differently by the evaluation heuristic. Adding a tilt event 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 all 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**) achieves all the goals.

The parameter-binding repair strategy proposes replacing the assumption **Parameter-of(Tilt1, Theta, Theta1)** with the assumption that the **Theta** parameter is **Theta3**, together with the constraint that **Theta3** equals 7°. This constraint is determined by the regression, which indicates that the desired value of the **Theta** parameter is the difference between the desired value of **SS1.orientation@Tilt1.start + Theta1** (7°) and the predicted value of **SS1.orientation@Tilt1.start** (0°). The evaluation

[2] Another four changeable assumptions are located by tracing through the dependency structure that explains why the predicted orientation of **B2** is 17°.

heuristic determines that this repair is a solution, achieving all the goals of the problem.

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

The two **CWA-exists** assumptions can both be replaced by assumptions that an erosion event occurred that destroyed 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. For the assumption **Tilt1.end < Tilt2.start**, the reordering strategy does not succeed because the predicted value of **SS1.orientation@Tilt1.start** (0°) does not equal its desired value at **Tilt2.start** (12°); similarly for the assumption **Deposition1.end < Tilt1.start**.

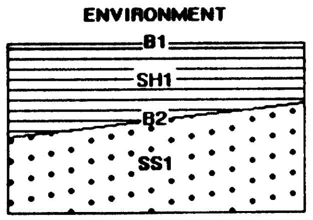
In sum, the debugger proposes seven potential repairs for this problem, of which three are considered solutions. Of the three solutions, the evaluation heuristic prefers the one in which the parameter of **Tilt1** is altered, since this produces an hypothesis with fewer events than the other two, both of which add new tilt events.

5.1.2 Double Deposition Example

This problem (Figure 52) is similar to that of the previous section, except that only one tilt event is needed to interpret the region. This example helps demonstrate both the efficiency of the generator and the robustness of the debugger. We ran the problem once using the full set of scenarios and once with no scenarios, forcing the debugger to solve the problem completely from scratch.

The generator solves the problem (Figure 52b) in essentially the same manner as described in Section 5.1.1, except that only one "tilted-sedimentary" scenario is used. The debugger's search for a solution is a bit lengthier. In fact, this is one of the more complicated geologic problems given to our current debugger — 10 unachieved goals (Figure 52c), a total of 95 assumptions examined, and 16 repairs proposed.

A. The Goal Diagram



B. One Plausible Solution

1. Deposition1(SS1, Sandstone)
2. Tilt1(9°)
3. Deposition2(SH1, Shale)

C. The Goal Propositions

1. Composition(SS1, Sandstone, Plan-end)
2. Composition(SH1, Shale, Plan-end)
3. Orientation(SS1, 9°, Plan-end)
4. Orientation(SH1, 0°, Plan-end)
5. Abuts(SS1, B2, Plan-end)
6. Abuts(SH1, B2, Plan-end)
7. Orientation(B2, 9°, Plan-end)
8. Abuts(Environment, B1, Plan-end)
9. Abuts(SH1, B1, Plan-end)
10. Orientation(B1, 0°, Plan-end)

Figure 52. Solving a Geologic Interpretation Problem Using the Debugger Alone

The debugger determines how the first goal proposition could have been achieved by hypothesizing that deposition of sandstone created **SS1**. The second goal likewise can be achieved by a deposition event creating **SH1** with shale composition. This produces a non-linear hypothesis, with the two deposition events unordered. The evaluation heuristic determines that both linearizations achieve the same goals — the two composition goals as well as the goal that the orientation of **SH1** is zero. The debugger fortuitously chooses to pursue the linearization where **SS1** is deposited first.

The next bug (unachieved goal) examined is that the orientation of **SS1** is zero, not 9°. A tilt event of 9° is proposed, that occurs sometime after the deposition of sandstone but unordered with respect to the shale deposition event. The debugger pursues the linearization where tilt precedes shale deposition, since the evaluation heuristic prefers it over the alternative ordering which introduces the new bug that the orientation of **SH1** is 9°, not 0°.

The debugger next tries to account for why **SS1** abuts **B2**. It makes five different suggestions: 1) **B2** is an erosional boundary and **SS1** was eroded, 2) **B2** is the lower depositional boundary of the sandstone formation, 3) **B2** is the upper boundary of the sandstone formation (the one abutting the environment after deposition), 4) **B2** is the lower depositional boundary of the shale formation, and 5) **B2** is a depositional boundary, formed by yet a third deposition event. The evaluation heuristic prefers the fourth hypothesis since that one also accounts for why **SH1** abuts **B2**, and why **B2** is oriented at 9°.

The next unachieved goal is **Abuts(Environment, B1, Plan-end)**. The debugger proposes four repairs similar to those above: 1) **B1** is an erosional boundary, 2) **B1** is the upper boundary of the sandstone formation, 3) **B1** is the upper boundary of the shale formation, and 4) **B1** is the upper boundary created by another deposition. The evaluation heuristic estimates that both the first and third repairs completely solve the problem, but #3 is preferred since it results in an hypothesis with fewer events. This hypothesis, which is the same as the one produced by the generator (Figure 52b), is then tested and found to be satisfactory.

In comparing the statistics for the generator and debugger (example G in Figure 47, and example J in Figure 48) we find that the generator produces a solution almost 40 times faster than the debugger. This can be attributed to two aspects of the GTD paradigm: 1) while each debugging step typically achieves just one goal, a single scenario can interpret multiple goals, and 2) the computational complexity of each debugging step is exponential, versus polynomial for the cost of instantiating a scenario. While in this example neither the generator nor the debugger deviates from the solution path, the debugger takes more steps (6 versus 2) and each is more expensive.

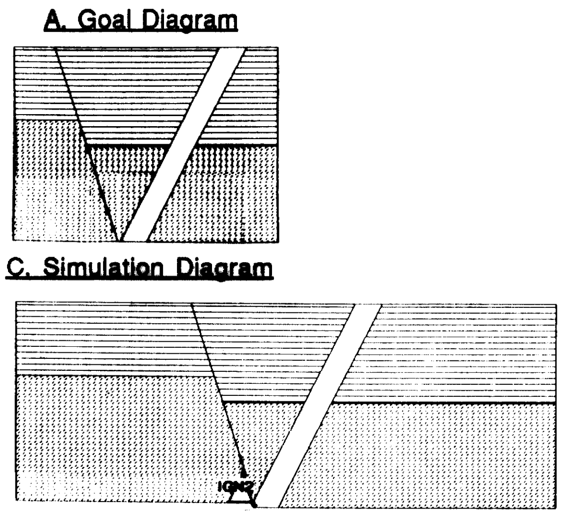
To explore the sensitivity of the debugger to perturbations in the input data, we ran the same example (with no scenarios in the library) except that we changed the orientation of the shale in the goal diagram from 0° to 9°, the same as the sandstone's. Somewhat surprisingly, the debugger proposes the same set of 16 repairs for both examples. The only difference is the way the evaluation heuristic rates the repairs.

This difference in ratings causes the debugger to pursue different hypotheses at two choice points. The first is when the tilt event is proposed — the debugger prefers the linearization where tilt follows the shale deposition, since this achieves the orientation goals for both **SS1** and **SH1**. The second difference is the choice at the very last step — the debugger prefers the proposal that **B1** is an erosional boundary since assuming that it is the upper depositional boundary of **SH1** implies that its orientation must be tilted at 9°. Thus, the debugger proposes the following hypothesis, which the tester verifies is indeed a valid solution:[3]

1. **Deposition1(SS1, Sandstone)**
2. **Deposition2(SH1, Shale)**
3. **Tilt1(9°)**
4. **Erosion1(B1)**

5.1.3 Extended Window Example With Incorrect Linearization

This section describes how the debugger can handle cases in which some linearizations of an hypothesis are not actually solutions and an incorrect ordering is chosen. In addition, this section and the next both illustrate how GORDIUS can handle bugs in which the goal and simulation diagrams differ topologically. This section illustrates how topological bugs can be repaired by replacing a temporal-ordering assumption; Section 5.1.4 illustrates replacing **CWA-exists** assumptions to fix topological bugs.



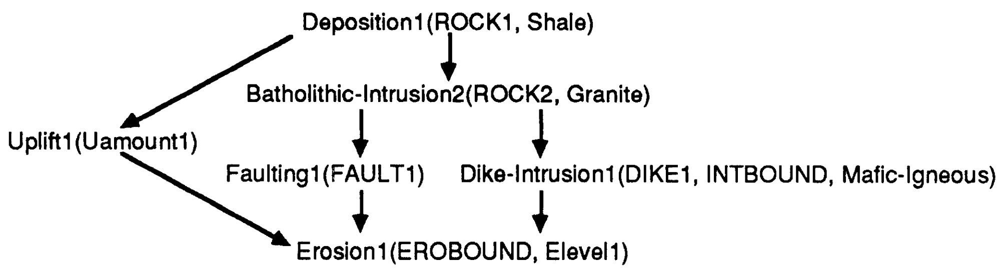
B. Linearized, Buggy Hypothesis

1. Deposition1(ROCK1, Shale)
2. Batholithic-Intrusion2(ROCK2, Granite)
3. Uplift1(Uamount1)
4. Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
5. Faulting1(FAULT1)
6. Erosion1(EROBOUND, Elevel1)

Figure 53. Incorrect Linearization Produces a Bug

The diagram of Figure 53a is similar to the example in Sections 2.1, 3.1 and 4.1, except that the bottom of the goal diagram window is deeper. Using the complete set of geologic scenarios (Appendix A), the initial hypothesis produced by the generator is:

[3] Uplift is not needed before the erosion since the tilt event is sufficient to raise the surface above sea-level.


Deposition1(ROCK1, Shale)
Batholithic-Intrusion2(ROCK2, Granite)
Uplift1(Uamount1)
Faulting1(FAULT1)
Dike-Intrusion1(DIKE1, INTBOUND, Mafic-Igneous)
Erosion1(EROBOUND, Elevel1)

To get a buggy hypothesis, we force the tester to choose a linearization in which **Dike-Intrusion1** precedes **Faulting1** (Figure 53b). In testing this hypothesis, the diagram matcher detects the bug that an extra piece of mafic-igneous appears within the window of the goal diagram (**IGN2** in Figure 53c).

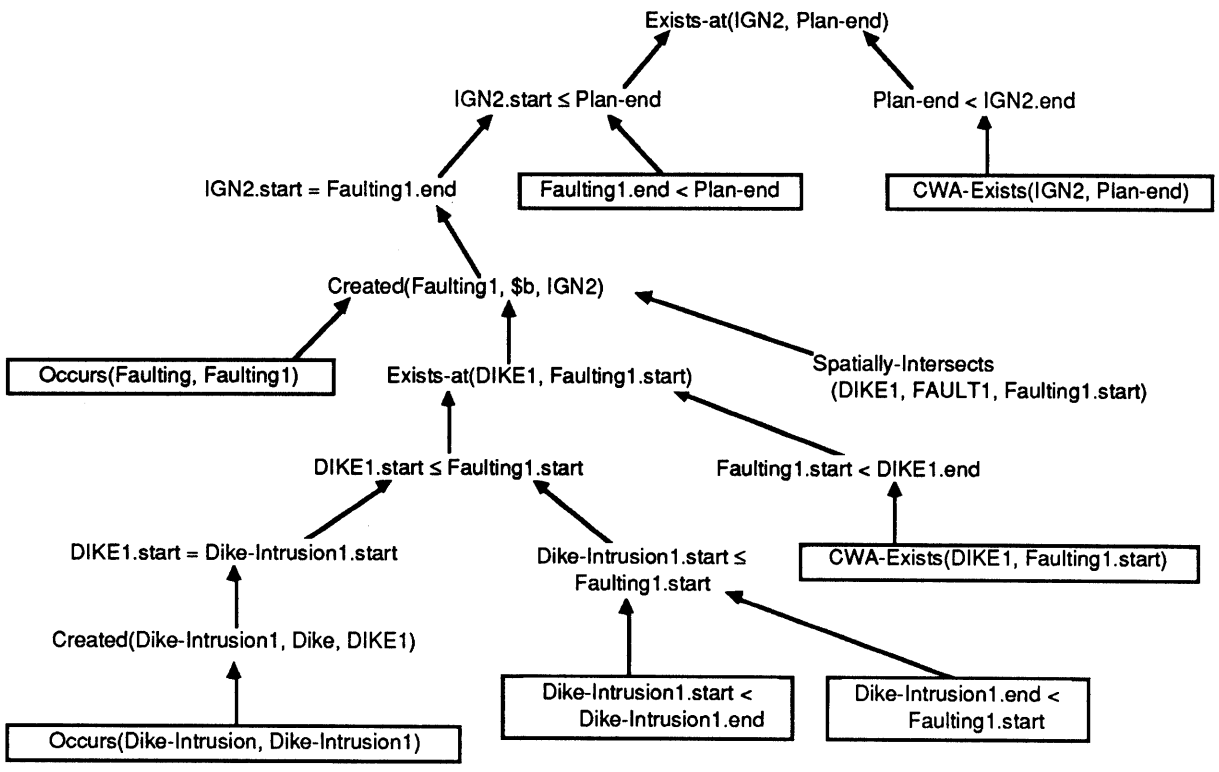

Exists-at(IGN2, Plan-end)
IGN2.start ≤ Plan-end
Plan-end < IGN2.end
IGN2.start = Faulting1.end
Faulting1.end < Plan-end
CWA-Exists(IGN2, Plan-end)
Created(Faulting1, $b, IGN2)
Occurs(Faulting, Faulting1)
Exists-at(DIKE1, Faulting1.start)
Spatially-Intersects (DIKE1, FAULT1, Faulting1.start)
DIKE1.start ≤ Faulting1.start
Faulting1.start < DIKE1.end
DIKE1.start = Dike-Intrusion1.start
Dike-Intrusion1.start ≤ Faulting1.start
CWA-Exists(DIKE1, Faulting1.start)
Created(Dike-Intrusion1, Dike, DIKE1)
Dike-Intrusion1.start < Dike-Intrusion1.end
Dike-Intrusion1.end < Faulting1.start
Occurs(Dike-Intrusion, Dike-Intrusion1)

Figure 54. Dependency Structure for the Bug that **IGN2** is an Extra Piece

The debugger examines all the assumptions in the dependency structure for this bug (Figure 54), except for two that are considered to be unchangeable (**Dike-Intrusion1.start < Dike-Intrusion1.end** and **Faulting1.end < Plan-end**). For the two **Occurs** assumptions, the debugger suggests deleting the faulting and dike-intrusion events. For the two **CWA-exists** assumptions, the debugger suggests adding an erosion event, together with the constraint that enough erosion occurs to destroy the existing object (**IGN2** or **DIKE1**).

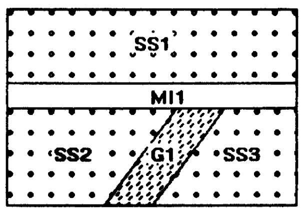
The assumption **Dike-Intrusion1.end < Faulting1.start** is the one added by the tester in order to linearize the hypothesis. The debugger proposes reordering the events so that faulting occurs before intrusion, which ensures that **IGN2** will not be created (i.e., **DIKE1** will not be split) since **DIKE1** will not exist at the time of the faulting.

The evaluation heuristic determines that this last modification is the only one that does not introduce any additional bugs. GORDIUS reorders the faulting and dike-intrusion events and tests the modified hypothesis. In this case, the simulation diagram does not contain any extra pieces of rock, so the hypothesis is accepted as a solution.

5.1.4 Double Intrusion Example

This interpretation problem (Figure 55) is an example where the *a priori* most plausible interpretation of a scenario pattern is not the correct one, due to unanticipated interactions between events. The problem also provides another instance of how the debugger can repair topological bugs.

A. Goal Diagram



B. Buggy Hypothesis

1. Deposition1(ROCK1, Sandstone)
   (SS1, SS2 and SS3 are
   pieces of ROCK1)
2. Dike-Intrusion1(G1, Granite)
3. Dike-Intrusion2
   (MI1, Mafic-Igneous)

C. Simulation Diagram

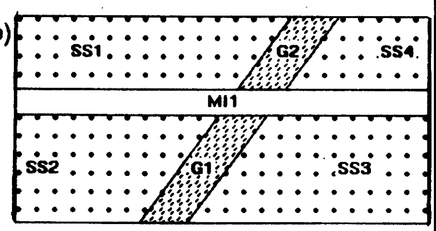


Figure 55. Unencapsulated Interactions in the Scenarios Produce a Buggy Hypothesis

The generator uses the "intrudes-through" scenario (R1|IGN|R2) to hypothesize that a **Rock-Creation1** event created formation **ROCK1,** followed by the **Dike-Intrusion1** event that intruded **G1** through **ROCK1**, splitting it into pieces **SS2** and **SS3**. Applying the scenario again, GORDIUS hypothesizes that **Rock-Creation2** created **ROCK2**, followed by **Dike-Intrusion2**, which intruded **MI1** and split **ROCK2** into **SS2** and **SS1**. Since both **ROCK1** and **ROCK2** contain the same piece (**SS2**), the unifier concludes that they are the same formation and, therefore, the two rock-creation events are really the same event.

Next, the generator hypothesizes that **Dike-Intrusion1** preceded **Dike-Intrusion2** using the "non-conformable-boundary" scenario ($\frac{R1}{R2 \mid R3}$), whose local interpretation indicates that **R1** is younger than both **R2** and **R3**. Finally, three applications of the "sedimentary-no-tilt" scenario are used to specialize the **Rock-Creation** to be a **Deposition** event.

Upon testing the initial hypothesis, the diagram matcher discovers that **G2** and **SS4** are two extra pieces in the simulation diagram (Figure 55).[4] The error stems from the fact that in reality **SS2** and **SS1** are not part of the same formation. While the generator relied on the presumption that the "intrudes-through" scenario provides the most plausible explanation for an igneous rock appearing between two rock-units of the same type, in this case it is merely coincidental that **SS2** and **SS1** are the same type.

The unencapsulated interaction in this example is that the splitting of the sandstone and granite formations by **MI1** are coupled — one cannot happen without the other. For the generator to handle this example correctly, a scenario should be added whose pattern is $\frac{\frac{R4}{IGN1}}{R2 \mid R3}$ and whose local interpretation indicates that **R2, R3** and **R4** are created by different events. With this scenario, the generator would produce the plausible solution:

1. **Deposition1(ROCK1, Sandstone)** — **SS2** and **SS3** are pieces of **ROCK1**
2. **Dike-Intrusion1(G1, Granite)**
3. **Dike-Intrusion2(MI1, Mafic-Igneous)**
4. **Uplift1(Uamount1)** — above sea-level
5. **Erosion1(Elevel1)** — down to the top of **MI1**
6. **Subsidence1(Samount1)** — below sea-level
7. **Deposition2(ROCK2, Sandstone)** — **SS1** is a piece of **ROCK2**

In the current case, the debugger tries to handle the problem that **G2** and **SS4** are extra pieces. Unfortunately, as described below, our current implementation only partly solves the problem. The debugger first proposes repairs to the bug that **G2** is an extra piece. The most promising modification is to replace the closed-world assumption that **G2** continues to exist at **Plan-end** by adding an

[4] Although it is equally likely that **SS1** is the extra piece, in this case the choice does not matter.

erosion event to destroy it. This introduces the bug that the erosion event cannot occur because its preconditions are not met. The debugger handles this case analogously to Section 4.1, proposing to add either an uplift or tilt event, but preferring to add uplift since it introduces no new bugs.

At this point, the evaluation heuristic estimates that the modified hypothesis solves the problem and it is sent to the tester. Actually, the evaluation heuristic determines that none of the goals are unachieved. It does not know for sure that all are in fact achieved, since from our causal models alone it cannot be determined whether the erosion will destroy **SS3** as well as **G2**.

In testing the hypothesis, in particular while determining parameter values for the diagrammatic simulation, the Quantity Lattice detects an inconsistency in the value of **Elevel1** (the level of erosion). The inconsistency occurs because the debugger constrains **Elevel1** to be less than the height of the bottom of **G2** (to destroy it), while the model of erosion constrains **Elevel1** to equal the height of the Earth's surface after erosion. The measured heights in the goal diagram of **G2** and the Earth's surface are inconsistent with these two constraints.

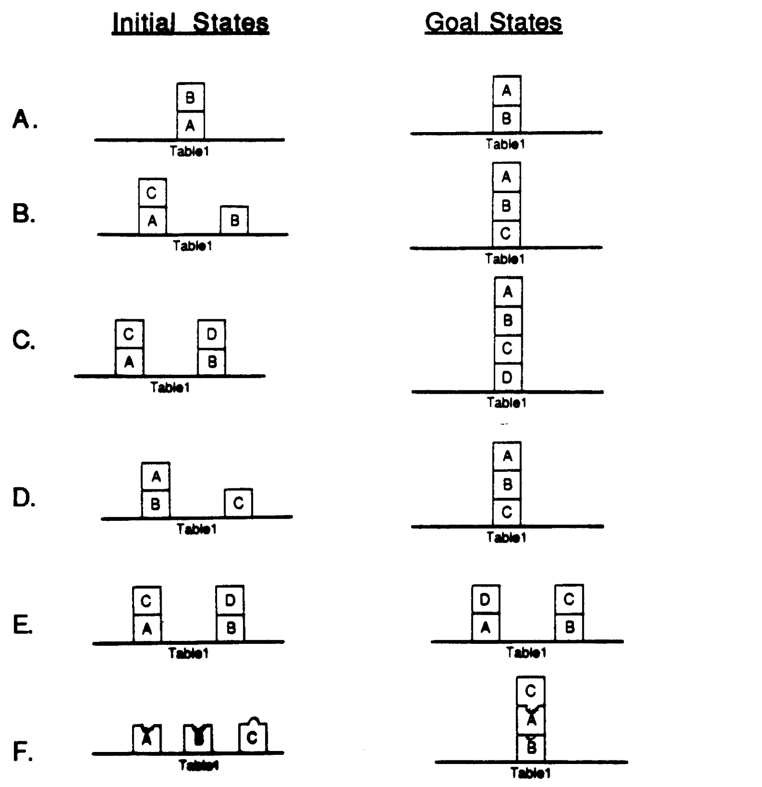
Given this bug manifestation, we would expect the debugger to analyze the bug's causal explanation to propose repairs. Unfortunately, as mentioned, GORDIUS fails to solve this example completely. In our current implementation the Quantity Lattice is not fully integrated with the rest of the system. Inconsistencies detected by the Quantity Lattice are not reported to the debugger, preventing GORDIUS from solving the problem.

While the problem in this case is an implementation failure, not a theory failure, it points up several requirements for systems, such as ours, that consist of several specialized representations. First, there needs to be a general method for handling inconsistencies. In our current implementation, each representation detects and handles inconsistencies independently; there is no general method for invoking the debugger when an inconsistency is found. Second, dependencies need to be represented in a consistent and unified manner. In our current implementation, some dependencies are actually computed *post hoc* when needed by the debugger. This strategy is not general enough in cases where the dependencies are needed in unforeseen circumstances, as in this example. We anticipate that a re-implementation of GORDIUS along these lines would be able to completely solve this example.

5.2 Blocks World Planning

In addition to geologic interpretation, we have tested GORDIUS using simple blocks-world planning problems. This domain is well studied in AI, and can serve as a testbed to compare our techniques with other problem-solving methods. In particular, we have found that GORDIUS compares favorably with other domain independent planners (e.g., [Sacerdoti], [Wilkins], [Chapman], [Vere]).

Figure 56 presents the initial states, goal states, and solutions produced by GORDIUS for the 6 blocks-world problems in [Sacerdoti]. To approximate the capabilities of NOAH, we had GORDIUS solve the problems without using any scenarios. Thus, the generator always produced the null hypothesis, and the bugs passed to the debugger were all goals not already achieved in the initial state.



Solutions

A.
1. Puton-Table1(B, Table1)
2. Puton2(A, B)

B.
1. Puton-Table1(C, Table1)
2. Puton2(B, C)
3. Puton3(A, B)

C.
1. Puton-Table1(D, Table1)
2. Puton2(C, D)
3. Puton3(B, C)
4. Puton4(A, B)

D.
1. Puton-Table1(A, Table1)
2. Puton2(B, C)
3. Puton3(A, B)

E.
1. Puton-Table1(D, Table1)
2. Puton2(C, B)
3. Puton3(D, A)

F.
1. Puton1(B, A)
2. Puton2(C, B)

Figure 56. Sample of Blocks-World Planning Problems Solved by GORDIUS Debugger (from [Sacerdoti])

Examples	A	B	C	D	E	F
1. Goal Statements	1	3	4	3	2	2
2. Changeable Assumptions	3	6	22	9	7	8
3. Repairs Suggested	3	6	16	7	7	6
4. Hypotheses Pursued	2	3	4	4	3	2
5. Solution Path Length	2	3	4	3	3	2
6. Efficiency (Ratio of 4 to 5)	1.0	1.0	1.0	1.3	1.0	1.0
7. Effort (Ratio of 3 to 1)	3.0	2.0	4.0	2.3	3.5	3.0
8. Time To Construct Hyp. (Min:Sec)	:05	1:23	5:06	:32	:28	:17
9. Time To Test Hypothesis	:03	:05	:09	:05	:04	:03

Figure 57. Statistics for Blocks-World Problems of Figure 56

Figure 57 presents the problem-solving statistics for these examples. The numbers are similar to those produced by the debugger in the geologic domain (Figure 48). In both domains, the "effort" expended averages about 3 repairs suggested for each goal achieved. This is somewhat surprising, as it was expected that more repairs would be suggested in the geologic domain, since each bug has more underlying assumptions (averaging about 9 versus 4 for the blocks-world domain) and there are more ways to affect geologic objects (the geologic domain has more types of events with many more effects per effect than does the blocks-world).

It turns out that the "effort" statistic is similar in both domains because in the geologic domain the repair strategies are applicable for a much smaller percentage of the underlying assumptions — 24% versus 82% in the blocks-world. In particular, fewer of the geologic event-occurrence and temporal-ordering assumptions can be replaced (by deleting and reordering events, respectively). The difference is because in the geologic domain bugs usually arise because a goal was never achieved in the first place, while in the blocks-world a bug is often caused by one event interfering with the positive effects of another event. Thus, deleting and reordering events are more applicable in the blocks-world because they eliminate the interference. This also suggests a debugging heuristic for the geologic domain — examine event-occurrence and temporal-ordering assumptions last.

The "efficiency" of the debugger in avoiding false paths is similar for both domains as well. Both domains exhibit nearly linear search behavior — GORDIUS deviates from the correct solution path in only one blocks-world problem, and in that case it pursues only one extra hypothesis. The false path is pursued because the correct solution involves undoing an already achieved goal and the debugger's goodness metric prefers hypotheses that minimize the number of unachieved goals.

Example C is somewhat of an anomaly, since it takes over five minutes to achieve only four goals. The problem is that the debugger evaluates several non-linear hypotheses in solving this example. Thus, a long run-time is to be expected since the evaluation is exponential for non-linear hypotheses. We believe, however, that the poor performance in this particular case reflects an as-yet-undiscovered bug in our implementation, since other problems in which non-linear hypotheses are evaluated (e.g., B and D) do not take nearly so long.

Although it is uncertain whether the good performance in solving blocks-world problems stems from the efficacy of our goodness metric or the simplicity of the domain, it is encouraging at least that our debugger alone can solve simple problems with some degree of efficiency.

5.3 Tower of Hanoi Problem

Tower of Hanoi problems were examined to confirm our expectations that the debugger's search strategy would exhibit poor performance in domains where the goals are very interdependent. This is the case in the Tower of Hanoi domain, where moving a ring from one post to another often interferes with the goal of moving other rings, and the correct solution often involves undoing previously achieved goals.

We had GORDIUS solve two-ring and three-ring Tower of Hanoi problems using the debugger alone (the causal models are presented in Appendix C). Figure 58 presents statistics for these problems. The effect of interference can be seen in both the efficiency and effort statistics. The debugger explores many more false paths and proposes more repairs for each goal (especially for the three-ring problem) than in either the geologic or blocks-world domains.

As noted above, this can be attributed to the goodness metric used — by preferring hypotheses with fewer unachieved goals the debugger explores paths that seem to be leading towards a solution, but are in fact dead ends. Even so, the focusing techniques of the debugger (combined with the simple search metric) enable GORDIUS to solve the problems exploring only a fraction of the possible search space. For the two-ring problem, for instance, GORDIUS suggests only 10% of the 84 possible plans of length 3 or less; for the three-ring problem, fewer than 1% of the possible plans are explored.

	Two Rings	Three Rings
1. Goal Statements	2	3
2. Changeable Assumptions	9	52
3. Repairs Suggested	7	36
4. Hypotheses Pursued	5	24
5. Solution Path Length	3	7
6. Efficiency (Ratio of 4 to 5)	1.7	3.4
7. Effort (Ratio of 3 to 1)	3.5	12.0
8. Time To Construct Hypothesis (Min:Sec)	:21	3:15
9. Time To Test Hypothesis	:05	:09

Figure 58. Statistics for the Tower of Hanoi Problems

5.4 Diagnosis in Semiconductor Fabrication Manufacturing

The semiconductor fabrication domain was explored primarily to exercise the tester and debugger in another complex, real-world domain. In our experiments, the tester's causal simulator was used to simulate the creation of electronic components on a silicon wafer (Figure 59). Simulation is complex in this domain both because of the large number of events simulated (typically around fifty) and because most events have several tens of effects that can potentially change the state of the wafer. The complexity is evident in the dependency structures produced — many have on the order of 500-1,000 nodes.

The diagnostic task we explored involved finding the parameters of events that could plausibly account for inconsistencies between the predicted and measured characteristics of some component (e.g., the resistance of a resistor). The diagnostic algorithm described in [Mohammed & Simmons] analyzes the dependency structures produced by our causal simulator to find the parameter-binding assumptions underlying a measurement's predicted value. The algorithm then determines constraints on the value of the parameter that could account for the measured value.

For example, possible causes for the resistance of resistor **A** in Figure 59 being too high include 1) the duration of the epitaxial-growth step was too short, and 2) the temperature of the diffusion step was too high (both of which would cause layer **SL1** to be thinner than predicted, thus increasing resistance).

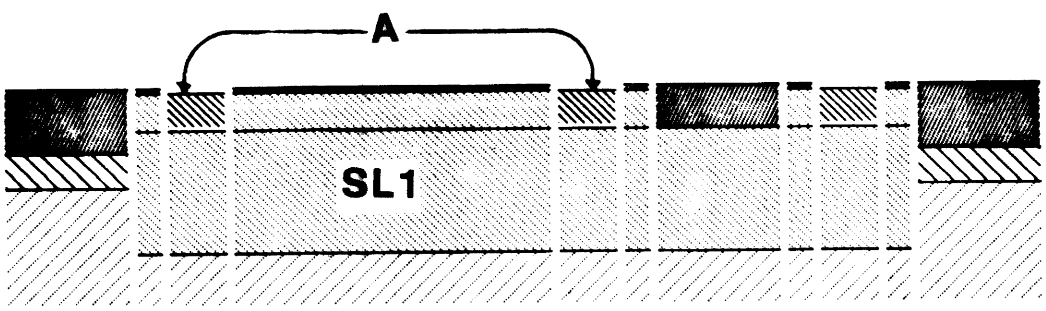


Figure 59. Simulation of the Manufacture of Two Resistors

Two minor modifications to the GORDIUS debugger are needed to achieve the same diagnostic functionality as reported in [Mohammed & Simmons]. First, the diagnostic algorithm considers only event parameters as the source of manufacturing faults. To replicate this behavior, we merely justify all assumptions other than parameter-bindings with the belief **given**, since the debugger treats assumptions with that degree of belief as unchangeable premises.

The second modification enhances the debugger's ability to do regression by enabling it to regress constraints through monotonically increasing and decreasing functions. The addition was needed because many of our semiconductor event models include user-defined functions which do not have associated definitions. For example, the function **Diff(Temperature, Duration)** represents the amount the thickness of a layer decreases due to diffusion. The function's actual definition is very complicated, and for most purposes it is sufficient to declare that **Diff** is monotonically increasing in each of its arguments.

To determine the relationship between parameter values and the measured characteristics of a component, the debugger needs to regress constraints through such functions. For example, the dependency structures encode that for layer **SL1** (simplifying a bit):

SL1.thickness@Plan-end =
Epi(Temperature1, Duration1) - Diff(Temperature2, Duration2),

where the function **Epi** represents the amount of epitaxial-growth and the function **Diff** is as described above. Algebraically solving for parameter **Temperature2** yields:

Diff-inverse$_X$(Epi(Temperature1, Duration1) - SL1.thickness@Plan-end, Duration2).

The system uses this constraint to determine the relationship between the thickness of **SL1** and **Temperature2**. First, it uses the table in Figure 60a and the knowledge that **Diff** is monotonically increasing in its first argument to infer that **Diff-inverse$_X$** is also monotonically increasing in its first argument. Next, since it knows that arithmetic difference is monotonically decreasing in its second argument, it can use the table in Figure 60b to compose **Diff-inverse$_X$** and difference (-) to infer that **Temperature2** is a monotonically decreasing function of the thickness of **SL1**. Thus, if **SL1** is too thick, it may be because **Temperature2** was too low.

The ability to regress constraints through monotonic functions may be useful in debugging as well. Although the regression always produced sufficient symbolic constraints in the problems we examined, there are likely to be problems where the direction in which to change a parameter may provide sufficient information to repair a bug. For example, if the bug is that a sedimentary formation is too thick and the system knows only that its thickness is proportional to the duration of deposition, it may be useful for the debugger to propose decreasing the duration, even if it cannot determine by how much.

A. Functional Dependence for Inverting Functions

C = F(A, B)	**A = F-Inverse$_X$(C, B)**
F(M+, M+)	F-inverse$_X$(M+, M-)
F(M+, M-)	F-inverse$_X$(M+, M+)
F(M-, M+)	F-inverse$_X$(M-, M+)
F(M-, M-)	F-inverse$_X$(M-, M-)

B. Dependence for Composing Functions

F(M+)	G(M+)	⇒	F(G(M+))
F(M+)	G(M-)	⇒	F(G(M-))
F(M-)	G(M+)	⇒	F(G(M-))
F(M-)	G(M-)	⇒	F(G(M+))

F(M+, M-) indicates that the function's value is monotonically increasing with respect to its first argument and decreasing with respect to its second argument.

Figure 60. Tables for Reasoning About Monotonic Functions

6. Combining Associational and Causal Reasoning

Our thesis is that the Generate, Test and Debug paradigm exhibits both efficient and robust behavior through its combination of associational and causal reasoning techniques. This chapter explores the relationship between the reasoning techniques used and describes how GTD takes advantage of the strengths of each technique. We argue that the problem-solving characteristics of the reasoning techniques depend largely on the extent to which they represent and reason about interactions between events. We also argue that the associational scenarios used by the generator are heuristic abstractions of the causal models and reasoning techniques used by the debugger, and we present the outline of an algorithm for deriving scenarios from the results of debugging.

By our definition, associational reasoning solves problems by associating features of the problem with their solution. In contrast, causal reasoning solves problems by reasoning about the structure and behavior of objects over time. Simply put, associational reasoning solves problems by recognition, while causal reasoning solves them by analysis.

In GTD, the generator achieves efficiency by composing the local interpretations of scenarios without detailed checking for interactions. This is a reasonable strategy under the presumption that the scenarios are (nearly) independent of one another. The presumption of composability implies that combining the solutions to sub-problems produces an hypothesis that achieves all the goals of the sub-problems. With totally independent scenarios, correct solutions can be generated in time proportional to the number of goals in the problem. As we argued in Section 2.3, using scenarios that encapsulate common patterns of interaction helps ensure that the presumption of composability will usually hold, so that simply matching and composing scenarios will often generate correct hypotheses.

However, in most complex domains, such as geology, no set of totally independent associational rules can be developed, aside from the trivial set of exactly one rule per problem (i.e., each pattern comprises the complete goal and initial states, and the local interpretation is a complete solution). By presuming composability, then, the generator will produce incorrect hypotheses in cases where some combination of scenarios does in fact interact. These unexpected interactions are the source of much of the brittleness typically observed in associational systems. Since associational reasoning techniques do not handle interactions not explicitly encoded in the rules, they do not know when the rules' range of applicability has been exceeded.

Given the fact that scenarios are not totally independent, it might seem desirable to check for unexpected interactions when composing scenarios, since an unprofitable line of search may be terminated if it can be shown that the scenarios interfere. Unfortunately, checking for non-independence is computationally very expensive. Thus, a cost/benefit tradeoff exists between the desire to prune the search space as early as possible by detecting interference and the expense of detection. Our response to this tradeoff is to test validity only after a complete hypothesis is generated. To avoid having the generator pursue obviously incorrect hypotheses, however, methods are used to detect a limited class of inconsistent hypotheses, such as those with temporal cycles. To maintain efficiency, the generator's consistency checking methods are all computationally inexpensive, at most quadratic with respect to the number of hypothesized events.

Although the consistency checks used by the generator are able to detect many inconsistent hypotheses, the generator does not detect all potential interactions. In addition, the generator may fail in situations where it does not have any scenario that matches one of the problem's goals. In such cases, more robust reasoning techniques are needed to fall back on.

The causal reasoning techniques used by our debugger fit this bill. While the associational reasoning presumes that scenarios can be composed independently, causal reasoning takes the opposite stance by explicitly representing and reasoning about interactions. This is accomplished using an explicit model of how the world works, one which represents time, the effects of change, and the persistence of objects over time. The causal reasoning techniques use their knowledge of these models to analyze how hypothesized events affect the state of the world.

For example, in an effort to avoid interference interactions, the debugger reasons about how modifications to an hypothesis affect already achieved goals. The debugger can also find novel ways to achieve goals by finding cooperative interactions between events.

For example, by reasoning about the cumulative effect of changes, the debugger can hypothesize that a sequence of tilt events, rather than a single event, accounts for the orientation of a rock-unit. This ability to reason about different types of interactions and to discover interactions not encoded explicitly gives causal reasoning a greater potential range of applicability than associational reasoning.

One downside of causal reasoning is its high computational cost. For domain models, such as ours, that incorporate relative and conditional effects, detecting interactions between events is in general exponential in the number of events occurring [Chapman]. In addition, because it reasons at a lower level of detail, the causal reasoner tends to search larger, more richly connected, search spaces than does the associational reasoner.

In short, the extent to which associational and causal reasoning deal with interactions gives them nearly opposite characteristics — efficient but brittle versus robust but slow. These differing characteristics indicate that the reasoning techniques are best suited for different aspects of the problem-solving task. The GTD paradigm takes advantage of the different strengths of the techniques to achieve an overall system that exhibits a high degree of performance (efficiency) and competence (robustness). Associational reasoning is used first under the presumption that the scenarios are sufficiently independent to produce correct hypotheses most of the time. Causal reasoning is reserved to focus on those problems not handled correctly by the associational reasoner. The presumption here is that the generated hypotheses are nearly correct, so only a small amount of expensive causal reasoning will be needed. Otherwise, if the initial hypothesis were far from a solution, it might be less work to start from scratch than to modify the hypothesis into a solution.

Although our analysis indicates that a GTD-type paradigm is best implemented with an associational generator and causal debugger, other choices are conceivable. Rather than using causal algorithms, for example, the debugger can be associational, employing heuristics that deal with the problem at a lower level of detail than do the rules in the generator (e.g., [Murthy], [Marcus]). Alternatively, the generator can use causal reasoning. In fact, the GORDIUS generator does use some causal reasoning to unify events and to detect inconsistencies, and the debugger uses some associational reasoning to associate the attributes of objects with the change statements of events that affect the attributes. Overall, however, the generator and debugger rely on the use of associational and causal reasoning to achieve a high degree of efficiency and robustness, respectively.

A presumption underlying our combination of associational and causal reasoning is that it is easier to construct robust causal models than to construct a robust set of associational rules. Otherwise, it would be less work simply to develop a robust generate and test system. There is empirical evidence from our own work and others (e.g., [Koton]) that robust causal models are indeed often easier to construct than robust associational rules. We are currently working to characterize why and for which domains this presumption holds. One possible explanation is that combinatorially it is easier to construct a small set of event models and general rules for combining their effects than it is to construct a set of rules that encompasses all possible ways the events could interact.

Another possible explanation is that associational rules are typically derived either from experience or from domain models. Since in any reasonably complex domain one is unlikely to experience enough specific cases to span a large fraction of the domain, the bulk of the rule set must be derived from domain models. Thus, domain models are a pragmatic precursor to the rules — hence it is more work to construct robust associational rules precisely when experience alone cannot span most of the domain.

6.1 Relationship Between Scenarios and Causal Models

The associational scenarios used by GORDIUS provide an encapsulated abstraction of the models and causal reasoning techniques used by the debugger. This section examines the types of knowledge abstracted away by the scenarios. We argue that scenarios are both abductions of the causal models and are abstractions along several dimensions, including level of vocabulary, degree of effects encoded, and chains of causal reasoning. We also speculate on the abstraction process itself, presenting the outline of a technique to derive scenarios by analyzing the actions of the debugger.

The following analysis presumes that the associational rules used can be derived from causal models. This is not an unreasonable restriction, since all the scenarios in the domains we explored are derivable from our causal understanding of the domains. This does not imply that the scenarios were actually derived from the causal models, only that our causal knowledge of the domains is sufficient that they could be.

The analysis starts by noting that causal models can be used in conjunction with a reasoning technique such as causal simulation to predict future states given an initial state and a set of events. That is, causal models can be used to compute the mapping:

$$\textbf{Initial-State} \times \textbf{Events} \rightarrow \textbf{Final-State}.$$

Scenarios, on the other hand, map from partial descriptions of the initial and final states (the pattern) to a set of events (the local interpretation):

$$\textbf{Initial-State} \times \textbf{Final-State} \rightarrow \textbf{Events}.$$

A scenario represents the knowledge that, starting from the initial world state, the events can achieve the goals of the final state.

Scenarios represent a twofold transformation of the mapping provided by the causal models — they are both an abduction and an abstraction of the causal mapping. Abduction transforms valid statements of the form "**A** causes **B**" into heuristics of the form "if **B** is observed then **A** may have caused it." More specifically, from the causal rule "from the **Initial-State**, the **Events** cause **Final-State**," abduction produces the scenario "in the context of **Initial-State**, if the **Final-State** is observed then the **Events** may have occurred."

Abduction is only heuristic because it implicitly presumes that **A** is the only cause of **B**, which may not be true. In geology, for instance, there are at least two plausible interpretations for observing sedimentary rock on top of igneous — 1) a sedimentary formation was deposited on an existing igneous rock-unit, or 2) an igneous formation intruded into the sedimentary rock-unit.[1] Although in some cases the cause is truly ambiguous, additional information may often constrain which hypothesis is more plausible. Given the two interpretations above, for example, if the same sedimentary material is found on the other side of the igneous rock, it is more likely that the second interpretation is correct — that the igneous intruded through the older sedimentary rock. The generator's preference for scenarios with more specific patterns is a heuristic

[1] We can actually construct an infinite number of additional, albeit less plausible, interpretations by adding pairs of uplift and subsidence events.

attempt to handle situations where the abduction is not valid, that is, where there is more than one plausible interpretation.

In addition to being an abduction of the causal mapping, associational scenarios abstract the mapping along several dimensions. One such dimension involves shifting towards a more abstract level of vocabulary. For example, in the object/attribute language of our causal models the fact that rock-unit **ru** is adjacent to one side of boundary **bnd** is described by:

(ru $\in$ bnd.side-1@time) or (ru $\in$ bnd.side-2@time).

In contrast, in the scenario patterns the same information is encoded without disjunction using the proposition **Abuts(ru, bnd, time)**. Although this proposition is more succinct, it loses information about the structure of the domain, namely that boundaries have two distinct sides.

A more important type of abstraction is that scenarios, which encapsulate patterns of interaction, abstract away the effects of events that do not interact with one another. As a result, scenario patterns typically contain a small subset of the effects that can be predicted using the causal models. For example, the "intrudes-through" scenario, which deals with the interactions of a dike-intrusion and a single formation, ignores effects explicitly included in the causal models including how the intrusion affects other formations, how it changes the appearance of the surface of the Earth, etc. This abstraction makes scenario patterns easier to match against the goal state, due to the smaller number of effects encoded. On the other hand, it implies that the scenario patterns cannot be used to recognize all the effects of an event, leading to buggy hypotheses in situations where the missing effects are important.

Another important dimension of abstraction is that scenarios abstract away the chains of causal reasoning (e.g., simulation or search) used to compute the causal mapping (**Initial-State $\times$ Events $\rightarrow$ Final-State**). For example, it often involves a fair amount of search to find a set of events that achieves one goal without interfering with the achievement of other goals. By abstracting away this search into a single rule, associational reasoning can avoid the expense in similar situations. One consequence of this abstraction is a lack of flexibility, since the reasoning steps must be reconstructed in order to obtain the information (e.g. dependencies) needed to modify buggy hypotheses.

Another consequence of abstracting away chains of reasoning is that the scenarios are limited to situations in which the same domain models and same problem-solving task apply. For example, the rule that a "fork" is useful in chess abstracts knowledge about legal chess moves (the domain models) and an analysis of possible moves and counter-moves (the problem solving). In particular, a fork encapsulates the coupled interaction that it simultaneously achieves two "threat-to-capture" goals. It is useful under the presumptions that capturing pieces is good and that the opponent can deal with only one threat at a time. The presumptions are specific to the standard chess game, however, and would be useless if the object of the game were to lose one's pieces or if two moves at a time were allowed. Similarly, scenarios are limited in their general applicability by being tied to specific domain models and tasks.

Note that this abstraction process is not necessarily limited to two levels. Just as the generator ignores interactions that are handled by the debugger, so too the debugger ignores certain interactions (e.g., reasoning about events that simultaneously affect the same attribute of an object) that may be important in other domains or tasks. We envisage that a more complete problem solver will utilize a spectrum of models and reasoning techniques tuned to those models. The more detailed models will offer a greater robustness, due mainly to the increased types of interactions that their associated reasoning techniques can handle. The more abstract models offer greater computational efficiency, due to the encapsulation of interactions and presumptions of non-interference (e.g., the GORDIUS generator presumes that its scenarios are independent, and the tester and debugger both presume that no simultaneous, interacting events occur).

6.1.1 Deriving Scenarios from Causal Models

Given the close correspondence between the knowledge encoded in the scenarios and causal domain models, a natural extension of the GTD paradigm would be to derive new scenarios by analyzing potential interactions in the causal models. We present below a rough outline of an unimplemented technique for deriving new scenarios by analyzing the results of debugging a problem. Using the results of debugging is a good strategy because it focuses the learning on those common patterns of interaction that actually arise in the course of solving problems.

A problem in creating scenarios is to encapsulate just enough of the potential interactions to learn generally useful scenarios — fairly independent, but not too specific. Our proposed learning technique involves analyzing the dependency structures produced during the course of testing and debugging an hypothesis to find the extent of interactions responsible for a bug. The causal explanations for those interactions would be generalized using techniques based on work in explanation-based learning (e.g., [DeJong], [Mitchell]).

Our proposal is to reason explicitly about the interference, cooperative, and coupled interactions that underlie a bug, and to encapsulate those interactions to create new scenarios. Interference interactions can be found by analyzing the causal dependency structure produced by the tester, which explains why the bug occurred. The assumptions underlying this dependency structure represent those decisions made by the problem solver that interact to produce the bug. Cooperative interactions can be found by analyzing the dependency structure produced by the debugger's evaluation heuristic, which explains how effects of the debugged hypothesis interact to repair the bug. Coupled interactions can be found by tracing forward from assumptions in both dependency structures to find change statements that with the same conditions as those effects that interact to repair the bug.

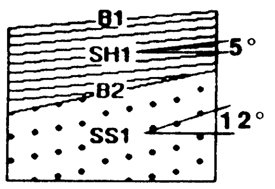
The goal state of the new scenario pattern can be determined by tracing forward from all interacting assumptions to find the top-level goals that they support. The scenario pattern's initial state would consist of those interacting assumptions that mention the time **Plan-start**. The local interpretation of the scenario would consist of all the event-occurrence, parameter-binding, and temporal-ordering assumptions that support the propositions in the scenario's goal state. The underlying closed-world assumptions can be ignored since they are equivalent to the presumption of independence used by the generator (i.e., no event outside the local interpretation interferes in the achievement of the scenario pattern). Finally, the scenario pattern and local interpretation can be generalized by replacing constants with variables using one of the algorithms discussed in the explanation-based learning literature (e.g., [DeJong], [Mitchell]).

As an example of how this learning technique might work in the domain of geologic interpretation, suppose GORDIUS has constructed the hypothesis of Figure 61b to interpret Figure 61a. The hypothesis is buggy because it predicts that **SS1** and **B2** have orientations of 17°, not 12°. Section 5.1.1 shows how the debugger solves the problem by altering the parameter of **Tilt1** from 12° to 7°.

The assumptions underlying the interference interactions in this example are that **Deposition1** occurs to create a horizontal rock-unit whose orientation is subsequently increased by the actions of **Tilt1** and **Tilt2**. The set of assumptions that cooperate to repair the bug is the same as the set of interfering assumptions, except for the assumption about the value of **Tilt1**'s parameter.

The set of top-level goals supported by the **Tilt1**, **Tilt2**, and **Deposition1** events are that **SS1** and **B2** are oriented at 12°, and **SH1** and **B1** are oriented at 5°. The generalized goal state pattern of the learned scenario would consist of two sedimentary rock-units (and their abutting boundaries) each oriented at a different angle. The initial state of the scenario pattern is simply the preconditions of **Deposition1**, that the Earth's surface is below sea-level. Finally, the local interpretation of the new scenario would consist of those assumptions underlying the causal explanations for the two orientation goals of the scenario pattern — that two depositions occur, each followed by a tilt.

A. Goal Diagram



B. Buggy Hypothesis	C. Debugged Solution
1. Deposition1(SS1, Sandstone)	1. Deposition1(SS1, Sandstone)
2. Tilt1(12°)	2. Tilt1(7°)
3. Deposition2(SH1, Shale)	3. Deposition2(SH1, Shale)
4. Tilt2(5°)	4. Tilt2(5°)

Figure 61. Deriving New Scenarios from the Results of Debugging

6.2 Search and Control Issues

A major problem in solving interpretation and planning problems is controlling the potentially explosive search. The combinatorics arise largely from interactions among events, especially where achieving one goal undoes other goals. Using nearly independent rules in the generator alleviates much of that interaction. However, control of search remains enough of a problem, especially for the debugger, to warrant careful consideration.

An important control issue is deciding the best way to search the space of hypotheses. Our approach is to perform best-first search, ordering hypotheses using a distance metric that prefers hypotheses with fewer unachieved goals and, secondarily, those with fewer events. Both the generator and debugger use this distance metric, although they differ in how the number of unachieved goals is calculated.

This distance metric is intuitively reasonable. For domains in which a solution consists of achieving all goals, the fewer the goals left unachieved, the closer one often is to a solution. Preferring a shorter hypothesis is reasonable for planning problems since shorter plans will presumably be easier to carry out. For interpretation problems, this preference follows Occam's razor that the simplest explanation is often the most likely. As described in Chapter 5, we have found this distance metric to be very effective in practice. Only in the Tower of Hanoi domain does the search deviate substantially from a correct solution path.

An important concern, especially for the generator, is the efficiency of calculating the distance metric. For both the generator and debugger, calculating the number of events in an hypothesis is trivial — it is simply the number of **Occurs** assumptions. Unfortunately, accurately calculating the number of unachieved goals is rather expensive. Thus, there is a tradeoff between accuracy and efficiency in calculating the distance metric. The algorithms used by the generator and debugger come down on different sides of this tradeoff — the generator emphasizes efficiency, while the debugger emphasizes accuracy (i.e., robustness).

The generator makes use of the presumption that scenarios are independent to efficiently estimate the number of unachieved goals. The generator simply presumes that a scenario achieves all the goals in its pattern, and that it does not undo any previously achieved goal. Thus, the number of unachieved goals remaining after a scenario is instantiated can be calculated in constant time for each goal.

The evaluation heuristic used by the debugger performs a more accurate determination by taking into account the possibility that a local bug repair can have global effects on the hypothesis. The algorithm, which updates closed-world assumptions invalidated by a bug repair, is exponential in the number of unordered events. Significant improvement in GORDIUS' performance would result from using more efficient evaluation methods to calculate interactions. For example, [Dean & Boddy] present a polynomial-time, albeit incomplete, algorithm to predict the effects of events. Another alternative, employed by [Lansky], is to reduce the size of the exponent by augmenting a complete, exponential evaluation algorithm with presumptions that certain events are independent of one another.

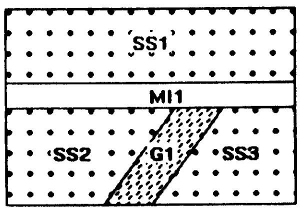
A more global control issue is deciding in which situations to use the generator, and when to resort to the debugger. Our current implementation employs the simple control strategy that the generator is used in isolation to construct an initial hypothesis, which is then tested. The expensive debugger remains as the "last line of defense" and is used only when the generator's scenarios are found to be interfering or incomplete (i.e., there are goals not matched by any scenario pattern). In the extreme case, with no scenarios in the generator's library, the debugger must solve the problem completely on its own. As the library of scenarios gets built up (perhaps using a learning algorithm, as in Section 6.1.1) the debugger will be used less and less, and the efficiency of the overall system will increase.

Although our current control strategy is designed to exploit the strengths of the various reasoning techniques used, it is not the only choice. Other control strategies that more fully integrate the generator and debugger may have certain advantages over the simple flow of control used in GORDIUS. Although we have not had time to do the experiments, different control strategies should be easy to implement in GORDIUS since the generator and debugger already use the same best-first search strategy and the same distance metric for ordering hypotheses. In our current implementation, in fact, all hypotheses are placed on a single priority queue, enabling GORDIUS to choose the hypothesis estimated to be closest to solving the problem, regardless of whether the hypothesis was constructed by the generator or debugger. Thus, if the debugger is found to be moving far from a solution, GORDIUS can try generating a new hypothesis.

With our current control strategy, however, only the debugger can make modifications to an hypothesis once it has been tested. This reflects the presumption that if the generator does not produce a correct solution initially, it does not have the knowledge necessary to correct errors it made. This presumption, however, is not always true. Given the incorrect hypothesis in Figure 62b, for instance, the debugger fixes the bugs that **G2** and **SS4** are extra pieces by inserting an uplift and erosion event after **Intrusion2** to completely erode away **G2** and **SS4**. However, since this repair also erodes away **SS1** it introduces additional bugs, namely all goals that mention **SS1** are no longer achieved. At this point, GORDIUS could continue with another series of debugging steps, but it would be much more efficient to use the generator since the "sedimentary-over-igneous" scenario, which hypothesizes erosion down to **MI1** followed by subsidence and deposition of **SS1**, matches all of the unachieved goals.

The computational savings in using the generator at this stage of the problem is substantial. This example points up the fact that research remains to develop problem-solving strategies that are more flexible than the static Generate-Test-Debug control strategy presented in this report.

A. Goal Diagram



B. Buggy Hypothesis

1. Deposition1(ROCK1, Sandstone)
2. Dike-Intrusion1(G1, Granite)
3. Dike-Intrusion2
   (MI1, Mafic-Igneous)

C. Simulation Diagram

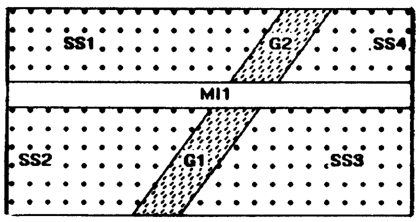


Figure 62. Example Showing Usefulness of Integrating Generator and Debugger

One promising control strategy is to use the generator first for all problems — both for achieving goals and for repairing bugs — and to use the debugger only if the generator does not have a suitable scenario. An advantage of this strategy is the potential for increased efficiency in solving problems in which a large amount of debugging is needed This potential must be weighed, however, against the fact that the scenario matcher will be invoked far more often. Until experiments are done, it is unclear which costs will predominate over the range of problems likely to be encountered. We believe, however, that the proposed control strategy may prove effective, especially given its similarities to the SOAR architecture [Laird], for which good performance has been demonstrated using the strategy of first trying to solve a subgoal at one level of representation and, only if unsuccessful, dropping down to a more detailed level.

6.3 Guidelines for Domains in Which GTD may be Useful

In this section, we develop guidelines for domains in which GTD may be a useful paradigm. The guidelines are based on our understanding of the relationship between the types of reasoning used in the generator and the debugger.

One important guideline is that the goals of the problem should not be totally independent. More precisely, there should not be a finite number of totally independent scenarios that cover all of the domain. If there were such a set, the presumption of composability embodied in the generator would always hold, and the generator would always produce a correct hypothesis. In such domains, the debugger would never be needed.

This guideline stipulates that the set of totally independent scenarios should not be finite because in theory one could cover any domain with an infinite set of independent scenarios, each of whose pattern is the complete initial and goal states of a problem, and whose local interpretation is the complete solution to the problem. The approach of having one scenario per problem is clearly intractable, however, especially on a serial computer where the cost of matching scenario patterns increases with the number of scenarios. Thus, there is a tradeoff between the desire to create a set of totally independent scenarios and the cost of determining which scenarios are applicable. In the geologic domain, where there are only a relatively few common scenarios but many infrequently occurring ones, the decision was made explicitly to maintain a small scenario library and to leave the uncommon cases for the debugger.

At the other extreme, an equally important guideline is that the goals of the problem should not be totally interdependent. If they were, the presumption of composability would never hold and the generator would always produce incorrect hypotheses. In such cases, especially when the generated hypothesis is far from a solution, the debugger may work harder than if it had solved the problem from scratch, without using the generator at all. For example, suppose we have a scenario that indicates how to solve a two ring Tower of Hanoi problem — move the top ring to the third post; move the next ring to the second post; move the top ring to the second post. Using that scenario in solving a three ring problem actually produces a very bad hypothesis — the debugger would have to change quite a few of the decisions made by the generator in order to get onto the correct solution path.

Most domains fall somewhere between the extremes of total independence and total interdependence of the goals. The ideal domain for GTD is one in which the goals are nearly independent, so that the debugger is used as infrequently as possible. For example, planning out assembly or machining tasks may be good domains for GTD because, although there are interactions, they tend to be localized and thus can be easily encapsulated into a small number of nearly independent scenarios.

While the above two guidelines address aspects of the domain itself, the guidelines below address aspects of our knowledge of the domain and the technology available for representing and reasoning about that knowledge.

One requirement to ensure efficiency is that we must identify a set of (nearly) independent scenarios that covers a large portion of the problem domain. If the scenarios interact to a large extent, or do not cover most of the domain, the debugger will have to be used frequently, resulting in a relatively inefficient system. Just knowing that the goals of the domain are nearly independent is not enough — we must be able to create the appropriate set of encapsulations. In principle, the set of scenarios may be constructed using a learning algorithm like the one described in Section 6.1.1, but this is not yet a proven technology. With current technology, we must still rely on people to identify the set of scenarios, although our analysis of scenarios as encapsulations of common patterns of interaction should help to provide a clear means to recognize good scenarios.

Two additional guidelines bear on achieving robust behavior by the debugger. First, we, as people, must have an understanding of the causality in the domain. Second, we must be able to represent the causality in a form that can be reasoned about by the system.

Although we believe that our representation language is sufficient for domains with discrete event models, there are many domains that need more robust models of how the world works. Extending the range of causal models that can be reasoned about is an active area of research in the qualitative physics community (e.g., [deKleer], [Forbus], [Williams]).

The particular debugging repair strategies described in this report are based on the models of causality, hypothesis construction, and the problem-solving task that underlie the domains we explored. To use the debugger in a domain based on different models, one must uncover the assumptions underlying the models and develop domain-independent repair strategies for each type of assumption. To use our debugging algorithm to diagnose electronic circuits, for instance, the model of hypothesis construction might have to include assumptions about the working status of components, while the model of causality might include closed-world assumptions concerning the topology of circuits (e.g., that there are no shorts or bridges).

A guideline for the tester is that it must be able to provide causal explanations for any bug detected, and it must be accurate over the range of problems likely to be encountered. In particular, the tester should avoid both false positives and false negatives. This condition is stronger than the assumption made by our debugger's model of causality that the domain models are complete and correct, since this implies only that the debugger avoids false negatives (i.e., rejecting valid solutions). It is also stronger than the presumption of composability underlying the generator. Thus, the models used by the tester must be at least as strong in predictive power as those used by either the generator or the debugger.

Note that the above are just guidelines. Problems that fall outside the guidelines, such as Tower of Hanoi, may still be solved within the GTD paradigm but the efficiency, accuracy, or robustness of the overall system will suffer if some of the guidelines are not met.

7. Related and Future Work

7.1 Associational Reasoning

Our approach of composing scenarios to generate correct, or nearly correct, hypotheses derives from work in cliché-based problem solving in which initial hypotheses are generated using a library of common operations (e.g. [Sussman], [Rich]). Our generator is also similar to case-based approaches (e.g. [Hammond], [Schank]) that index into previously solved problems to find examples that nearly match the current situation. Together with our generator, these approaches share the twin problems that 1) matching the associations (scenarios/clichés/cases) is expensive, and 2) it is not always easy to choose which of the matching associations to pursue. Since these problems have not been the focus of our research, we have chosen the relatively simple solutions of matching based on a rete-net algorithm and choosing scenarios based on the specificity ordering of their patterns. Other work, however, has focused more carefully on these problems, particularly on how to match efficiently (e.g., [Wills], [Kolodner]).

The efficiency gained by using heuristics that take interactions into account has been noted by other researchers. Many of these efforts, like our own, deal with interactions using heuristic rules that essentially pre-compile the interactions inherent in solving particular sets of goals (e.g., [Rich], [C. Hayes]). Another approach is to reason explicitly about potential goal interactions before solving the problem (e.g., [Hammond], [Wilensky]). Although this approach tends to be more flexible in finding unanticipated interactions, it is also more expensive due to the extra reasoning steps needed. In the domains explored, we have not found this extra flexibility to be necessary.

The presumption that the associational rules are (nearly) independent is present in other systems, as well. The "linearity assumption" used in HACKER is an assumption that the goals of the problem can be solved independently (see [Sussman], p. 53). The assumption of rule independence is often found, implicitly at least, in systems that use associational if-then rules with certainty factors (e.g., [Shortliffe], [Duda]) In describing Mycin, for instance, [Shortliffe] indicates:

"since the rules are not explicitly related to one another ... modifications and additions of new rules need not require complex considerations regarding interactions with the remainder of the system's knowledge." (p. 165)

Unfortunately, this view is not completely accurate since, as we have argued, in many domains it is often impossible to find totally independent rules. Analysis of the certainty-factor model used by Mycin shows that, at best, the rules can be presumed to be only conditionally independent with respect to the current rule base [Heckerman]. Although the presumption of independence is used by many different associational reasoning systems, the point is that it is just a heuristic and the problem solver needs to rely on some type of non-associational reasoning to handle unanticipated interactions.

The rule-based system R1 [J. McDermott], whose task is computer component layout, provides an interesting case study in the strengths and weaknesses of the type of match and compose approach used by our generator. While straightforward match and compose suffices for most of R1's subtasks, a backtracking technique is needed for laying out the Unibus. Our conjecture is that while most of the layout task can be factored into independent subtasks,[1] the interactions in the Unibus slot assignment task are too extensive to permit the decomposition into independent rules needed to make match and compose an effective technique.

7.2 Causal Simulation

Many researchers have investigated different methods and representations for doing causal simulation. The thrust of this section is to indicate the sources for our models of events, time, and change.

Our event representation derives from the precondition/action representations of domain-independent planners (e.g., [Fikes], [Sacerdoti], [Wilkins]). We have had to extend the basic action representation, however, to represent and reason about the geology and semiconductor fabrication domains. In particular, our models represent relative effects (e.g., an attribute increases or decreases by a certain amount), conditional effects, quantified effects, and the creation and destruction of objects. While many of these features

[1] More precisely, the task can be factored into subtasks that can be performed sequentially.

are also found in the formal planning model of [Pednault], that model has not yet been implemented.

One significant departure from traditional work on action representation is our explicit use of time and persistence. The explicit representation of time enables us to make inter-temporal comparisons (e.g., "if the salary in 1984 is greater than the salary in 1987 then ..."), which we have found greatly facilitates the representation of complex events. The explicit representation of persistence enables us to handle the frame problem [Hayes, 73], since the notion of persistence embodies the frame axiom that an attribute does not change unless some event affects it. Persistence also simplifies our action representations by enabling us to eliminate the "delete" list of STRIPS-type operators. Our models do not have to encode explicitly which values get "deleted" since by definition a change to an attribute ends the persistence of its current value.

Our temporal representations have been influenced by the work of Drew McDermott and students (e.g., [McDermott], [Dean], [Shoham]). In particular, that research calls for an explicit representation of persistence and argues that one can understand the effect of events as changing the persistence of statements.

A difference from our models is in the types of things that persist. In our work, attributes of objects persist; in the work of McDermott, et. al., it is propositions that persist over time. Although the approaches are formally very similar, pragmatically there are differences. For one thing, using histories of attributes[2] embodies the implicit presumption that an attribute can only have one value at a time. That presumption enables the problem solver to efficiently discover value contradictions, such as the fact that a rock-unit cannot be both igneous and sedimentary at the same time. On the other hand, using propositions as primitives is more flexible, particularly for representing binary relations between objects, such as "greater than." We prefer our history-based representation because the ability to detect contradictions efficiently is valuable for doing debugging.

Our causal simulator shares much of the emphasis found in work on qualitative simulation (e.g., [deKleer], [Forbus], [Kuipers]) that the

[2] The idea of histories derives from [Hayes, 85].

simulator should be able to make predictions without complete, quantitative knowledge of the situation, and that the simulation should provide causal explanations for the changes that occur. The main difference is in emphasis — rather than exploring the limits of qualitative reasoning, our work stresses the integration of qualitative and quantitative information, and the use of multiple representations of knowledge, such as quantities, sets, and diagrams.

7.3 Debugging and Domain-Independent Planning

Our debugging 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 & Williams]) and algorithmic debugging [Shapiro]. Our work extends these approaches by providing principled repair strategies that determine how to replace the underlying assumptions once they have been located. For example, the algorithmic debugger of [Shapiro] has no automatic way of replacing faulty assumptions, relying on the user to supply the necessary bug fixes once the underlying assumptions have been found.

Our assumption-oriented debugging approach stands in contrast to work in which bug repair heuristics are associated either with bug manifestations (e.g., [Alterman], [Marcus]) or with certain stereotypical causal explanations (e.g., [Hammond], [Sussman]). By "stereotypical explanations" we mean that the debuggers have a predetermined catalog of explanations for bug types, and they try to match the explanations for bugs against these stereotypical explanations. Both [Hammond] and [Sussman] accomplish this by asking a predetermined set of questions and using the answers to index into a library of bug types, such as "Prerequisite Clobbers Brother Goal" (PCBG) or "Desired-Effect:Blocked-Precondition" (DE:BP). Associated with each bug type are one or more repair strategies.

The problem with these approaches is that they are not very robust. In particular, they cannot handle bugs arising from unforeseen combinations of assumptions that do not match any of the catalogued bug types. In contrast, our approach handles the large number of possible ways bugs can arise by decomposing them into combinations of a small set of underlying assumptions. Since we do not have to anticipate all possible patterns of assumptions that can lead to bug manifestations, our assumption-oriented approach tends both to give greater coverage and to suggest more alternative repairs than other approaches.

To support the claim of greater coverage, we analyzed the bug types of [Sussman] and [Hammond] and found that our implemented debugger can duplicate all four of the repair strategies in [Sussman] and all but two of the 17 repairs in [Hammond]. The two bug types not handled are ones where simultaneous, interacting events are needed to repair the bug, which is beyond our current representation language. By extending our model of causality, our debugging approach should handle all of the repairs suggested in [Hammond].

In addition to covering more bug types, our debugging approach provides more flexibility by suggesting alternative repairs for the same bug type. As an example, consider the PCBG bug type in [Sussman] that occurs when an event **A**, in attempting to achieve the preconditions of an event **B**, undoes a goal that had been achieved by event **C** (see Figure 63). The only repair for this bug type given in [Sussman] is to reorder events **A** and **C**. The DE:BP bug type in [Hammond], which covers similar situations, has an additional repair strategy of replacing event **B** with one that does not have the offending precondition. Our debugger would suggest even more repairs, including inserting an event to reachieve the goal, replacing event **A**, and changing **A**'s parameters so as to make the goal and precondition true simultaneously.

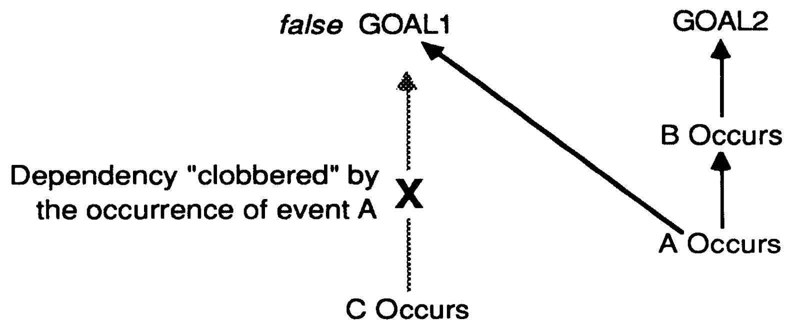


Figure 63. Schematic Causal Explanation for the "Prerequisite Clobbers Brother Goal" Bug

Our analysis of the debugger's relative completeness is carried out in terms of the assumptions underlying problem-solving models. A similar analysis is also found in [Goldstein], where the bugs that can arise in simple LOGO programs are categorized in terms of common mistakes that occur during the construction of the program. Little attempt is made, however, to catalog a complete set of planning errors for the domain studied, and no attempt is made to relate the bugs to faulty assumptions made about the model of causality used.

The basic approach to debugging described in this report — repair one bug at a time by analyzing domain models, and then evaluate how the local repair affects the hypothesis as a whole — is similar to the approach used by most domain-independent planners (e.g. [Sacerdoti], [Wilkins], [Tate], [Vere], [Chapman]). In fact, all our blocks-world planning examples were run without the generator, in effect "planning by debugging a blank sheet of paper." A major difference, however, is that most domain-independent planners use hypothesis *refinement*, while our debugger uses hypothesis *transformation*. In the refinement approach, the system can only add information to its current hypothesis, making the plan increasingly more detailed. In our transformational approach, information may be deleted as well, removing previous decisions such as which events occur or how parameters are bound.

Phrased in terms of possible worlds, in the refinement approach each modified hypothesis defines a set of possible worlds that is a subset of the possible worlds defined by the original hypothesis. In the transformational approach, the set of possible worlds defined by a modified hypothesis may overlap, or even be disjoint from, the possible worlds defined by the original hypothesis. One implication of this analysis is that the transformational approach is not guaranteed to converge on a solution; thus control issues are particularly important with this approach.

An advantage of the transformational approach to planning is the increased flexibility it gives the planner. In particular, in the refinement approach previous decisions can be removed only by backtracking — typically the control strategy is to continually try to refine and only backtrack when a contradiction is reached. In the transformational approach, the planner does not have to choose between refining the hypothesis and backtracking, since both options are essentially treated the same: some of our transformational repair strategies add information (which is equivalent to refinement) and some remove information (which is equivalent to backtracking).

The refinement approach tries to avoid backtracking by making decisions only when the problem forces them to be made. The difficulty is that such a least-commitment strategy often produces hypotheses with many unordered events and under-constrained parameter bindings, making it very expensive to determine which goals are achieved and when bugs exist. We have found that a reasonable computational approach is to make commitments early that seem likely to lead to a correct solution but to be prepared to remove those commitments if later problem solving dictates. For example, if a bug repair strategy proposes a non-linear sequence of events, the debugger pursues the linearization that minimizes the number of unachieved goals, even though the ordering chosen might not prove to be the right one in the end. If the ordering turns out to be wrong, the debugger can change it using the "temporal-ordering" repair strategy.

The transformational approach is also particularly useful for systems that use specialized planning algorithms, such as route-planners, to produce initial hypotheses, and then use a domain-independent planner to fine tune the plans. If bugs are discovered in the hypothesis produced by the specialized planner, the domain-independent planner cannot backtrack since it does not have a trace of the choice points made by the specialized algorithm. In such cases, transformational operators are needed.

7.4 Future Work

We believe that GTD, with its integration of different reasoning techniques, has applicability in a wide range of domains. We intend to use the GTD paradigm for solving problems in other complex planning and interpretation domains, such as route planning and economic interpretation. It is likely that these new domains will require extensions of the current implementation. In particular, new quantitative simulation algorithms will have to be developed to supplement the current diagrammatic simulator, which is fairly dependent on the geologic domain.

Unlike in geologic interpretation, in route planning and economic interpretation the initial and goal states are not limited to single points in time. For example, the initial state of a route planning problem might be "robot is at home at **Plan-start**, and the bank is open from 10AM to 3PM" and the goal state might be "robot should be at MIT at **Plan-end**, and it should be at the concert at 8PM." While the current debugger can already handle problems of this sort, the generator would need to be extended to handle goals involving multiple time points. A potential problem here is determining which propositions of the scenario patterns hold without resorting to a full-blown simulation. For example, it might be expensive to determine whether the scenario pattern "robot is at **X** at **t1**, and bank is open at **t1**" matches the initial state above (in particular, whether the robot being at home persists until 10AM when the bank opens).

We also believe that GTD-type paradigms have applicability over a variety of problem types, including diagnosis and design. There are some indications of that from our work in diagnosing manufacturing faults in semiconductor fabrication [Mohammed], [Simmons & Mohammed]. Extending GORDIUS to handle other types of problems will probably necessitate additional bug repair strategies, since the assumptions included in the models of the problem-solving task and hypothesis construction will likely be different from those in the models used for interpretation and planning tasks. A fruitful line of research might be the development of a language for expressing such models, from which the system might derive the underlying assumptions for itself.

The most notable assumptions that not currently handled by our debugger are the assumptions that the domain models are correct and complete. A bug repair strategy to handle these closed-world assumptions (i.e., to do theory formation) would be somewhat different from the current set of strategies. While the current strategies all evaluate a repair with respect to the current problem, repairs that change domain models should only be made with respect to a set of problems. The change should not only fix the current problem but should be consistent with the solutions to all previous problems.

A natural extension of the GTD paradigm is to derive new associational rules based on the results of debugging a problem. A step in that direction is the technique suggested in Section 6.1.1 for creating encapsulations by reasoning about the extent of interactions in a problem. There remains much work, however, to implement such a technique and to determine whether it is, in fact, useful for determining what to encapsulate and which interactions to ignore.

A more radical extension to the basic paradigm is to use more than just two levels of reasoning, where each level makes fewer simplifying presumptions and handles more types of interactions than the previous level. The more detailed levels would be used only when the less detailed ones failed to solve a particular problem. For example, one could have a system that combines associational reasoning, discrete causal reasoning, and reasoning about continuous process models in which interactions between simultaneous events can be represented. [Simmons & Mohammed] presents an example of how such a proposal might work in the semiconductor fabrication domain.

8. Conclusions

This report has explored the combination of associational and causal reasoning techniques to achieve both efficient and robust problem-solving behavior. The Generate, Test and Debug paradigm and its implementation, a program called GORDIUS, were developed to take advantage of the strengths, and compensate for the weaknesses, of associational and causal reasoning. GTD integrates multiple reasoning techniques, where each technique employs representations and inference algorithms tuned to its particular task and requirements.

Associational reasoning is used to efficiently generate initial hypotheses. The generator uses heuristic rules, called scenarios, that associate patterns in the input with sets of events that could plausibly achieve the patterns. Each scenario represents a fragment of the solution — the generator forms complete solutions by composing events from different matching scenarios. In composing events, the generator tries to unify together those that really signify the same event. The unifying technique is based on general principles about the world, such as that an object can be created in only one way, by only one event.

The tester uses detailed simulation to verify hypotheses. The tester used in GORDIUS relies on a largely qualitative, domain-independent causal simulation. The causal simulator updates a world model to reflect changes to the attributes of objects. It records causal dependencies that indicate how events affect the world, and how objects and attributes persist over time. To increase the accuracy of the test for geologic interpretation problems, the causal simulator is augmented with a quantitative, diagrammatic simulator that constructs a sequence of diagrams to represent the spatial effects of events. GORDIUS compares diagrams using a general-purpose diagram matcher that finds topological and geometrical correspondences between parts of the diagrams.

Much of our research effort has involved developing and implementing a domain-independent theory of debugging plans and interpretations. The theory is based on the premise that all bugs stem from faulty assumptions made during hypothesis construction and testing. Bugs are repaired by replacing faulty assumptions. The debugger uses three causal reasoning techniques to determine which assumptions to replace and how to replace them: 1) it locates the assumptions underlying bugs by tracing back through the dependency structures produced by the tester, 2) it indicates the direction in which to change assumptions by regressing desired values back through the dependencies, and 3) it repairs bugs using domain-independent repair strategies that determine which assumptions to add or delete based on analyses of the dependencies, regressed values, and causal domain models.

A fourth causal reasoning technique is used to determine the global impact of a bug repair by estimating the number of remaining bugs. The evaluation heuristic is based on the tester's causal simulator and reasons about the interactions between events caused by a bug repair. The algorithm achieves some measure of efficiently by focusing on the closed-world assumptions that become invalid as a result of the repair.

In practice, our experiments indicate that the six repair strategies implemented in GORDIUS can handle a wide range of bugs in several different domains, arising from a number of different combinations of assumptions. More formally, the coverage provided by our current debugger can be characterized by examining the premises contained in three different models upon which GORDIUS is built — models of causality, hypothesis construction, and the problem-solving task. The model of causality indicates that bugs depend only on assumptions about the events that occur, the initial and goal states, and various closed-world assumptions, including assumptions about the persistence of attributes and objects. The model of hypothesis construction indicates that the events can be represented by assumptions about the types of the events, their parameter bindings, and orderings between events. The task model indicates that one cannot replace assumptions about the initial and goal states, since that would be tantamount to solving a different problem.

Our analysis of the relationship between associational and causal reasoning leads to the conclusion that the problem-solving characteristics of the generator and debugger depend on the extent to which interactions between events are represented and reasoned about. The generator's efficiency stems from using scenarios that encapsulate common patterns of interaction, and from presuming that scenarios can be composed independently. The correctness of the hypotheses generated depends on the extent to which this presumption of composability holds.

The robustness of the debugger derives, in large part, from its ability to reason causally about interactions using domain models that explicitly represent time, persistence, and the effects of events. Since reasoning about interactions is computationally expensive, however, we prefer to use the causal reasoning sparingly, reserving it to incrementally modify incorrect hypotheses produced by the generator.

As an indication of the close relationship between associational rules and causal models, we have outlined a technique to derive associational rules from the results of debugging an hypothesis. We believe that GTD provides a natural foundation for studying such explanation-based learning. In particular, the bugs detected provide a focus for what to learn, reasoning about interactions provides the scope for how much to learn, and learning within a problem-solving framework provides the researcher an opportunity to evaluate the usefulness of what has been learned.

Guidelines for other domains in which GTD may be useful were developed based on our understanding of the characteristics and roles of the different reasoning techniques. The ideal domain is one in which the goals to be achieved are neither totally independent nor totally interdependent, with overall efficiency increasing as the goals become more nearly independent. For the generator, we must be able to identify a set of (nearly) independent scenarios that span a large portion of the domain. An accurate testing algorithm must be available, in particular, one that avoids false negatives. Finally, the debugger requires causal domain models and an ability to construct causal explanations for bugs, otherwise the debugging algorithms may perform no better than random permutation of the hypothesis.

We believe that the GTD paradigm has applicability in other interpretation and planning domains, such as economic interpretation and route planning, and in other types of problems, such as design and diagnosis. Important areas of research include extending the debugger to handle different types of assumptions that arise from the use of different underlying models, and adding more flexible control strategies to decide dynamically when to use the different reasoning techniques.

This report has demonstrated how efficient and robust problem-solving behavior can be achieved through a combination of reasoning techniques that differ largely in how they treat interactions. Throughout this research, we have attempted to analyze why the reasoning techniques behave as they do, and to ferret out the important problem-solving issues and presumptions behind the GTD paradigm. Both the specific reasoning techniques developed and our analyses of those techniques should provide a firm foundation for applying GTD to a wide variety of tasks.

9. References

R. Alterman, An Adaptive Planner, Proceedings of AAAI-86, Philadephia, PA, 1986.

A. Barr, E. Feigenbaum, *The Handbook of Artificial Intelligence, Volume 1*, William Kaufman, Los Altos, CA, 1981.

B. Baumgart, Geometric Modelling for Computer Vision, AI Memo 249, Stanford, 1974.

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

R. Davis, Diagnostic Reasoning Based on Structure and Behavior, *Artificial Intelligence*, vol. 24, pp. 347-410, 1984.

T. Dean, D. McDermott, Temporal Data Base Management, *Artificial Intelligence*, vol. 32, no. 1, pp. 1-55, 1987.

T. Dean, M. Boddy, Incremental Causal Reasoning, Proceedings of AAAI-87, Seattle, WA, 1987.

G. DeJong, R. Mooney, Explanation-Based Learning: An Alternative Approach, *Machine Learning 1*, 1986.

J. deKleer, J. S. Brown, A Qualitative Physics Based on Confluences, *Artificial Intelligence*, vol. 24, pp. 7-84, 1984.

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

R. Duda, P. Hart, et. al., Development of the PROSPECTOR Consultation System for Mineral Exploration, SRI Technical Report, October 1978.

R. Fikes, N. Nilsson, STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence*, vol. 2, pp. 189-208, 1971.

K. Forbus, Qualitative Process Theory, *Artificial Intelligence*, vol. 24, pp. 85-168, 1984.

C. Forgy, RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, *Artificial Intelligence*, vol. 19, pp. 17-38, 1982.

I. Goldstein, Summary of MYCROFT: A System for Understanding Simple Picture Programs, *Artificial Intelligence*, vol. 6, no. 3, 1975.

K. Hammond, Case-based Planning: An Integrated Theory of Planning, Learning and Memory, PhD thesis, RR 488, Yale University, 1986. Also, Explaining and Repairing Plans that Fail, Proceedings of Tenth IJCAI, Milan, Italy, 1987.

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

D. Heckerman, E. Horvitz, The Myth of Modularity in Rule-Based Systems, KSL Memo 86-33, Stanford, 1986.

C. Hayes, Using Goal Interactions to Guide Planning, Proceedings of AAAI-87, Seattle, WA, 1987.

P. Hayes, The Frame Problem and Related Problems in Artificial Intelligence, in A. Elithorn and D. Jones (eds.) *Artificial Intelligence and Human Thinking*, Elsevier, Holland, 1973.

P. Hayes, The Second Naive Physics Manifesto, in J. R. Hobbs and R. C. Moore (eds.) *Formal Theories of the Commonsense World*, Ablex Publishing, Norwood, NJ, 1985.

J. Kolodner, Reconstructive Memory: A Computer Model, *Cognitive Science*, vol. 7, pp. 281-328, 1983.

P. Koton, Empirical and Model-Based Reasoning in Expert Systems, Proceedings of Ninth IJCAI, Los Angeles, CA, 1985.

P. Koton, Using Experience in Learning and Problem Solving, PhD thesis, Laboratory of Computer Science, MIT, 1988.

B. Kuipers, Commonsense Reasoning about Causality: Deriving Behavior from Structure, *Artificial Intelligence*, vol. 24, pp. 169-204, 1984.

J. Laird, A. Newell, P. Rosenbloom, SOAR: An Architecture for General Intelligence, *Artificial Intelligence*, vol. 33, no. 1, pp. 1-64, 1987.

A. Lansky, D. Fogelsong, Localized Representation and Planning Methods for Parallel Domains, Proceedings of AAAI-87, Seattle, WA, 1987.

The Mathlab Group, MACSYMA Reference Manual, Laboratory of Computer Science, MIT, 1977.

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

D. McAllester, The Use of Equality in Deduction and Knowledge Representation, AI Technical Report 550, MIT, 1980.

D. McDermott, A Temporal Logic for Reasoning About Processes and Plans, *Cognitive Science*, vol. 6, pp. 101-155, 1982.

J. McDermott, R1: A Rule-Based Configurer of Computer Systems, *Artificial Intelligence*, vol. 19, no. 1, pp. 39-88, 1982.

T. Mitchell, R. Keller, S. Kedar-Cabelli, Explanation-Based Generalization: A Unifying Approach, *Machine Learning 1*, 1986.

J. Mohammed, R. Simmons, Qualitative Simulation of Semiconductor Fabrication, Proceedings of AAAI-86, Philadelphia, PA, 1986. See also below, R. Simmons & J. Mohammed.

S. Murthy, S. Addanki, PROMPT: An Innovative Design Tool, Proceedings of AAAI-87, Seattle, WA, 1987.

R. Patil, Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis, LCS Technical Report 267, MIT, 1981.

E. Pednault, Preliminary Report on a Theory of Plan Synthesis, Technical Report 358, AI Center, SRI International, 1985.

C. Rich, D. Waters, Abstraction, Inspection and Debugging in Programming, AI Memo 634, MIT, 1981.

C. Rich, Inspection Methods in Programming, AI Memo 1005, MIT; also to appear in *Artificial Intelligence*, 1988.

E. Sacerdoti, *A Structure for Plans and Behavior*, American Elsevier, New York, NY, 1977.

E. Sacks, Qualitative Mathematical Reasoning, Proceedings of Ninth IJCAI, Los Angeles, CA, 1985.

R. Schank, *Dynamic Memory: A Theory of Learning in Computers and People*, Cambridge University Press, 1982.

E. Shapiro, *Algorithmic Program Debugging*, MIT Press, Cambridge, MA, 1982.

Y. Shoham, *Reasoning About Change: Time and Causation from the Standpoint of Artificial Intelligence*, MIT Press, Cambridge, MA, 1988.

E. Shortliffe, *Computer Based Medical Consultations: MYCIN*, American Elsevier, New York, NY, 1976.

R. Simmons, R. Davis, Representations for Reasoning About Change, AI Memo 702, MIT, 1983.

R. Simmons, Representing and Reasoning About Change in Geologic Interpretation, AI Technical Report 749, MIT, 1983.

R. Simmons, 'Commonsense' Arithmetic Reasoning, Proceedings of AAAI-86, Philadelphia, PA, 1986.

R. Simmons, J. Mohammed, Causal Modeling of Semiconductor Manufacture, SPAR Technical Report 65, 1987.

H. Simon, *The Sciences of the Artificial*, MIT Press, Cambridge, MA, 1969.

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

G. Sussman, *A Computer Model of Skill Acquisition*, American Elsevier, New York, NY, 1977.

A. Tate, Generating Project Networks, Proceedings of Fifth IJCAI, Cambridge, MA 1977.

J. Van Baalen, R. Davis, Overview of a Theory of Representation Design, Proceedings of AAAI-88, Minneapolis, MN, 1988.

S. Vere, Splicing Plans to Achieve Misordered Goals, Proceedings of Ninth IJCAI, Los Angeles, CA, 1985.

M. Wellman, R. Simmons, Mechanisms for Reasoning About Sets, Proceedings of AAAI-88, Minneapolis, MN, 1988.

R. Wilensky, *Planning and Understanding*, Addison-Wesley, Reading, MA, 1983.

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

B. Williams, Qualitative Analysis of MOS Circuits, *Artificial Intelligence*, vol. 24, pp. 281-346, 1984.

L. Wills, Automated Program Recognition, AI Technical Report 904, MIT, 1986.

Appendix A. Geologic Interpretation Scenarios

This appendix illustrates both schematically and propositionally the fifteen scenarios, and their associated definitions, used to solve the geologic interpretation problems presented in this report.

1. Primitive Predicates

The following are the low-level predicates used to construct scenario patterns. Each predicate has an associated procedure that knows how to examine the goal state to determine whether the predicate holds given a set of argument bindings (arguments are preceded by a '?') . If one or more of the arguments of the predicate is uninstantiated, the associated procedure returns a binding list of all objects that complete the predicate.

1. Is-Igneous-Material(?material) — is the **material** an igneous type
2. Is-Sedimentary-Material(?material) — is the **material** a sedimentary type
3. Is-Environment(?env) — is **env** the **Environment** object

   The following predicates all have a temporal argument **?t**, meaning that the predicate is defined to hold at time **t**.
4. Is-Rock-Unit(?r, ?t) — is object **r** an existing rock-unit at time **t**
5. Is-Boundary(?b, ?t) — is object **b** an existing boundary at time **t**
6. Abuts(?r, ?b, ?t) — does rock-unit **r** abut boundary **b** at time **t**
7. Connected(?b1, ?b2, ?t) — do **b1** and **b2** share a common end-point at time **t**
8. Orientation-Equal?(?obj, ?theta, ?t) — does the orientation of object **obj** (approximately) equal angle **theta**
9. Below(?r, ?b, ?t) — is rock-unit **r** below boundary **b**
10. Above(?r, ?b, ?t) — is rock-unit **r** above boundary **b**
11. RL-Side-Of(?r, ?b, ?rl-side, ?t) — is rock-unit **r** on the right or left side of boundary **b**
12. Between(?b1, ?r1, ?b2, ?t) — does rock **r1** fall between boundaries **b1** and **b2**
13. Linear(?b1, ?b2, ?t) — do boundaries **b1** and **b2** lie (approximately) along the same line

2. Defined Predicates

The following are the intermediate-level predicates that have been found useful in defining scenario patterns for the geologic domain. Each defined predicate is a conjunction of primitive predicates and/or other defined predicates.

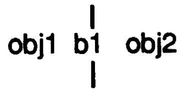
1. Same-Type(?r1, ?r2, ?mat, ?t) — do **r1** and **r2** have the same composition
   a. ?r1.material@?t = ?mat
   b. ?r2.material@?t = ?mat

2. Is-Igneous(?r, ?mat, ?t) — is the material composition of **r** igneous
   a. ?r.material@?t = ?mat
   b. Is-Igneous-Material(?mat)

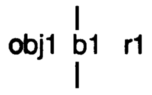
3. Is-Sedimentary(?r, ?mat, ?t) — is the material composition of **r** sedimentary
   a. ?r.material@?t = ?mat
   b. Is-Sedimentary-Material(?mat)

4. Oriented-Boundary(?b, ?theta, ?t) — is **b** a boundary orientated at **theta** at time **t**
   a. Is-Boundary(?b, ?t)
   b. Orientation-Equal(?b, ?theta, ?t)

5. Shared-Boundary(?obj1, ?b1, ?obj2, ?t) — do **obj1** and **obj2** share a common boundary **b1** (an object is either a rock-unit or the **Environment**)
   a. Abuts(?obj1, ?b1, ?t)
   b. Abuts(?obj2, ?b1, ?t)



6. Shared-Rock-Boundary(?obj1, ?b1, ?r1, ?t)
   a. Abuts(?obj1, ?b1, ?t)
   b. Abuts(?r1, ?b1, ?t)
   c. Is-Rock-Unit(?r1, ?t)

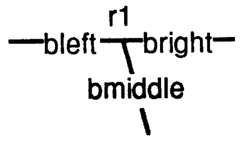


7. Parallel(?b1, ?b2, ?theta, ?t) — do boundaries **b1** and **b2** have (approximately) the same orientation **theta**
   a. Oriented-Boundary(?b1, ?theta, ?t)
   b. Oriented-Boundary(?b2, ?theta, ?t)

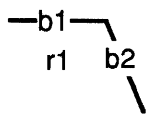
8. Colinear(?b1, ?b2, ?theta, ?t) — are boundaries **b1** and **b2** colinear
   a. Oriented-Boundary(?b1, ?theta, ?t)
   b. Oriented-Boundary(?b2, ?theta, ?t)
   c. Linear(?b1, ?b2, ?t)

9. Colinear-Connected(?b1, ?b2, ?theta, ?t) — are boundaries **b1** and **b2** colinear and have an endpoint in common
   a. Oriented-Boundary(?b1, ?theta, ?t)
   b. Oriented-Boundary(?b2, ?theta, ?t)
   c. Connected(?b1, ?b2, ?t)

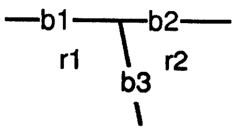
10. T-Joint(?r1, ?bleft, ?bmiddle, ?bright, ?theta, ?t)
   a. Colinear-Connected(?bleft, ?bright, ?theta, ?t)
   b. Abuts(?r1, ?bleft, ?t)
   c. Abuts(?r1, ?bright, ?t)
   d. Connected(?bleft, ?bmiddle, ?t)
   e. Connected(?bright, ?bmiddle, ?t)



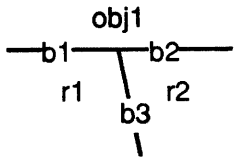
11. L-Joint(?b1, ?r1, ?b2, ?t)
   a. Abuts(?r1, ?b1, ?t)
   b. Abuts(?r1, ?b2, ?t)
   c. Connected(?b1, ?b2, ?t)



12. Colinear-Rock-Boundary(?b1, ?b2, ?theta, ?r1, ?b3, ?r2, ?t)
   a. Is-Rock-Unit(?r1, ?t)
   b. Shared-Rock-Boundary(?r1, ?b3, ?r2, ?t)
   c. L-Joint(?b1, ?r1, ?b3, ?t)
   d. L-Joint(?b2, ?r2, ?b3, ?t)
   e. Colinear-Connected(?b1, ?b2, ?theta, ?t)



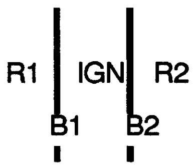
13. Faced-Rock-Boundary(?obj1, ?b1, ?b2, ?theta, ?b3, ?r2, ?t)
   a. Colinear-Rock-Boundary(?b1, ?b2, ?theta, ?r1, ?b3, ?r2, ?t)
   b. Shared-Rock-Boundary(?obj1, ?b1, ?r1, ?t)
   c. Shared-Rock-Boundary(?obj1, ?b2, ?r2, ?t)

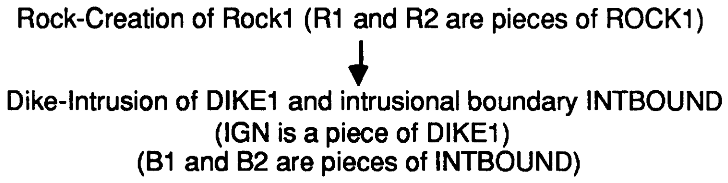


3. Scenarios

The following presents graphical representations of the scenario patterns and local interpretations as well as the declarative representations actually used by the generator.

1. INTRUDES-THROUGH




Rock-Creation of Rock1 (R1 and R2 are pieces of ROCK1)
Dike-Intrusion of DIKE1 and intrusional boundary INTBOUND
(IGN is a piece of DIKE1)
(B1 and B2 are pieces of INTBOUND)

Scenario Pattern:
   Is-Igneous(?ign, ?ign-material, ?t)
   Shared-Rock-Boundary(?ign, ?b1, ?r1, ?t)
   Shared-Rock-Boundary(?ign, ?b2, ?r2, ?t)
   Same-Type(?r1, ?r2, ?rock-material, ?t)
   Parallel(?b1, ?b2, ?theta, ?t)
   Between(?b1, ?ign, ?b2, ?t)
   Symmetries: (?r1, ?r2), (?b1, ?b2)

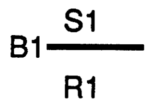
Local Interpretation:
   Occurs(Rock-Creation, ?rock-creation1)
   Occurs(Dike-Intrusion, ?dike-intrusion1)

   Parameter-of(?rock-creation1, Rock, ?rock1)
   Parameter-of(?dike-intrusion1, Rock, ?dike1)
   Parameter-of(?dike-intrusion1, Boundary, ?intbound)
   Parameter-of(?dike-intrusion1, Icomposition, ?ign-material)
   Spatially-Intersects(?rock1, ?dike1, ?dike-intrusion1.start)

   ?rock-creation1.end $\leq$?dike-intrusion1.start

   Piece-of(?ign, ?dike1)
   Piece-of(?r1, ?rock1)   Piece-of(?r2, ?rock1)
   Piece-of(?b1, ?intbound)   Piece-of(?b2, ?intbound)

2. SEDIMENTARY-OVER-ROCK



$$\begin{tikzcd}\text{Rock-Creation of Rock1 (R1 is piece of ROCK1)} \arrow[d] \\ \text{Deposition of SED1 (S1 is a piece of SED1)}\end{tikzcd}$$

Scenario Pattern:
Is-Sedimentary(?s1, ?material, ?t)
Shared-Rock-Boundary(?s1, ?b1, ?r1, ?t)
Above(?s1, ?b1, ?t)
Orientation-Equal(?b1, ?theta, ?t)

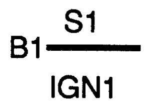
Local Interpretation:
Occurs(Rock-Creation, ?rock-creation1)
Occurs(Deposition, ?deposition1)

Parameter-of(?rock-creation1, Rock, ?rock1)
Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Boundary, ?intbound)
Parameter-of(?deposition1, Dcomposition, ?material)
Is-Deposited-Upon(?rock, ?deposition1.start)

?rock-creation1.end $\leq$?deposition1.start

Piece-of(?s1, ?sed1)
Piece-of(?r1, ?rock1)
Piece-of(?b1, ?depbound)

3. IGNEOUS-UNDER-SEDIMENTARY



$$\begin{tikzcd}\text{Deposition of SED1 (S1 is piece of SED1)} \arrow[d] \\ \text{Batholithic-Intrusion of BATH1 into SED1 (IGN is piece of BATH1)}\end{tikzcd}$$

Scenario Pattern:
Is-Igneous(?ign1, ?ign-material, ?t)
Shared-Rock-Boundary(?ign1, ?b1, ?s1, ?t)
Below(?ign1, ?b1, ?t)
Orientation-Equal(?b1, ?theta, ?t)

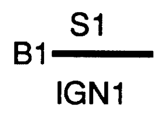
Local Interpretation:
Occurs(Deposition, ?deposition1)
Occurs(Batholithic-Intrusion, ?batholithic-intrusion1)

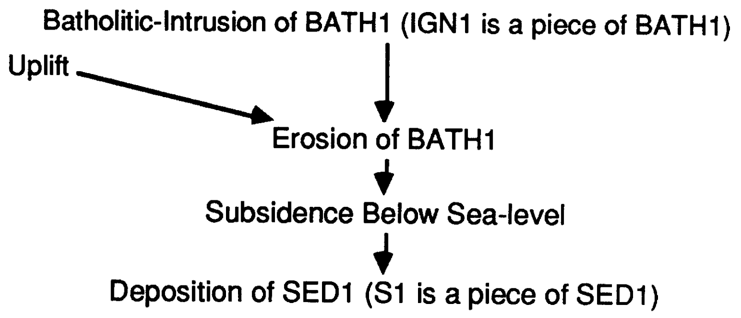
Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?batholithic-intrusion1, Rock, ?bath1)
Parameter-of(?batholithic-intrusion1, Boundary, ?intbound)
Parameter-of(?batholithic-intrusion1, Icomposition, ?ign-material)
Spatially-Intersects(?sed1, ?bath1, ?batholithic-intrusion1.start)

?deposition1.end $\leq$?batholithic-intrusion1.start

Piece-of(?ign1, ?bath1)
Piece-of(?s1, ?sed1)
Piece-of(?b1, ?intbound)

4. SEDIMENTARY-OVER-IGNEOUS




Batholitic-Intrusion of BATH1 (IGN1 is a piece of BATH1)
Uplift
Erosion of BATH1
Subsidence Below Sea-level
Deposition of SED1 (S1 is a piece of SED1)

Scenario Pattern:

```
Is-Sedimentary(?s1, ?sed-material, ?t)
Shared-Rock-Boundary(?s1, ?b1, ?ign1, ?t)
Above(?s1, ?b1, ?t)
Is-Igneous(?ign1, ?ign-material, ?t)
Orientation-Equal(?b1, ?theta, ?t)
```

Local Interpretation:

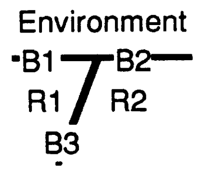
```
Occurs(Batholithic-Intrusion, ?batholithic-intrusion1)
Occurs(Uplift, ?uplift1)
Occurs(Erosion, ?erosion1)
Occurs(Subsidence, ?subsidence1)
Occurs(Deposition, ?deposition1)

Parameter-of(?batholithic-intrusion1, Rock, ?bath1)
Parameter-of(?batholithic-intrusion1, Icomposition, ?ign-material)
Parameter-of(?uplift1, Uamount, ?uamount1)
Parameter-of(?erosion1, Elevel, ?elevel1)
Parameter-of(?subsidence1, Samount, ?samount1)
Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Boundary, ?depbound)
Parameter-of(?deposition1, Dcomposition, ?sed-material)
?uamount1 ≥ Sea-Level – Surface.top.height@?uplift1.start
?elevel1 ≤ ?bath1.top.height@?erosion1.start
?samount1 > Surface.top.height@?subsidence1.start - Sea-Level
Is-Deposited-Upon(?bath1, ?deposition1.start)

?batholithic-intrusion1.end ≤ ?erosion1.start
?uplift1.end ≤ ?erosion1.start
?erosion1.end ≤ ?subsidence1.start
?subsidence1.end ≤ ?deposition1.start

Piece-of(?s1, ?sed1)
Piece-of(?b1, ?depbound)
Piece-of(?ign1, ?bath1)
```

5. EROSION-ENDS-BOUNDARY



$$\begin{tikzcd}[column sep=small]
\text{Boundary-Creation of BOUND3 (B3 is a piece of BOUND3)} \arrow[drr] & \text{Rock-Creation of ROCK2 (R2 is a piece of ROCK2)} \arrow[dr] & & \text{Rock-Creation of ROCK1 (R1 is a piece of ROCK1)} \arrow[dl] & \text{Uplift above Sea-level} \arrow[dll] \\
& & \text{Erosion creating erosional boundary EROBOUND (B1 and B2 are pieces of EROBOUND)} & &
\end{tikzcd}$$

Scenario Pattern:

```
Is-Environment(?env)
Faced-Rock-Boundary(?env, ?b1, ?b2, ?theta, ?r1, ?b3, ?r2, ?t)
Above(?env, ?b1, ?t)
```
Symmetries: (?r1, ?r2), (?b1, ?b2)

Local Interpretation:

```
Occurs(Rock-Creation, ?rock-creation1)
Occurs(Rock-Creation, ?rock-creation2)
Occurs(Boundary-Creation, ?boundary-creation1)
Occurs(Uplift, ?uplift1)
Occurs(Erosion, ?erosion1)

Parameter-of(?rock-creation1, Rock, ?rock1)
Parameter-of(?rock-creation2, Rock, ?rock2)
Parameter-of(?boundary-creation1, Boundary, ?bound1)
Parameter-of(?uplift1, Uamount, ?uamount1)
Parameter-of(?erosion1, Boundary, ?erobound)
Parameter-of(?erosion1, Elevel, ?elevel1)
?uamount1 ≥ Sea-Level – Surface.top.height@?uplift1.Start
?elevel1 ≤ ?rock1.top.height@?erosion1.start
?elevel1 ≤ ?rock2.top.height@?erosion1.start
?elevel1 ≤ ?bound1.top.height@?erosion1.start

?rock-creation1.end ≤ ?erosion1.start
?rock-creation2.end ≤ ?erosion1.start
?boundary-creation1.end ≤ ?erosion1.start
?uplift1.end ≤ ?erosion1.start

Piece-of(?r1, ?rock1)
Piece-of(?r2, ?rock2)
Piece-of(?b1, ?erobound)
Piece-of(?b2, ?erobound)
Piece-of(?b3, ?bound1)
```

6. ERODED-SEDIMENTARY

Environment

— B1 ————

S1

S1.orientation@t = B1.orientation@t

$$\begin{tikzcd}
\text{Deposition of SED1 (S1 is a piece of SED1)} \arrow[d] \\
\text{Uplift above Sea-level} \arrow[d] \\
\text{Erosion creating erosional boundary EROBOUND (B1 is a piece of EROBOUND)}
\end{tikzcd}$$

Scenario Pattern:

```
Is-Environment(?env)
Shared-Boundary(?s1, ?b1, ?env, ?t)
Above(?env, ?b1, ?t)
Is-Sedimentary(?s1, ?sed-material, ?t)
Orientation-Equal(?b1, ?theta, ?t)
Orientation-Equal(?s1, ?theta, ?t)
```

Local Interpretation:

```
Occurs(Deposition, ?deposition1)          Occurs(Uplift, ?uplift1)
Occurs(Erosion, ?erosion1)

Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Dcomposition, ?sed-material)
Parameter-of(?uplift1, Uamount, ?uamount1)
Parameter-of(?erosion1, Elevel, ?elevel1)
Parameter-of(?erosion1, Boundary, ?erobound)
?uamount1 ≥ Sea-Level – Surface.top.height@?uplift1.Start
?elevel1 ≤ ?sed1.top.height@?erosion1.start

?deposition1.end ≤ ?uplift1.start         ?uplift1.end ≤ ?erosion1.start

Piece-of(?s1, ?sed1)                      Piece-of(?b1, ?erobound)
```

7. SEDIMENTARY-ON-SURFACE

Environment

— B1 ————

S1

S1.orientation@t = B1.orientation@t

Deposition creating SED1 and depositional boundary DEPBOUND
(S1 is a piece of SED1)
(B1 is a piece of DEPBOUND)

Scenario Pattern:

```
Is-Environment(?env)
Shared-Boundary(?s1, ?b1, ?env, ?t)
Above(?env, ?b1, ?t)
Is-Sedimentary(?s1, ?sed-material, ?t)
Orientation-Equal(?b1, ?theta, ?t)
Orientation-Equal(?s1, ?theta, ?t)
```

Local Interpretation:

```
Occurs(Deposition, ?deposition1)

Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Upper-Boundary, ?depbound)
Parameter-of(?deposition1, Dcomposition, ?sed-material)

Piece-of(?s1, ?sed1)                      Piece-of(?b1, ?depbound)
```

8. TILTED-SEDIMENTARY

S1

S1.orientation@t ≠ 0

$$\begin{tikzcd}\text{Deposition of SED1 (S1 is a piece of SED1)} \arrow[d] \\ \text{Tilt by angle of Orientation(S1)}\end{tikzcd}$$

Scenario Pattern:

Is-Sedimentary(?s1, ?sed-material, ?t)
Orientation-Equal(?s1, ?theta1, ?t)
?theta1 ≠ 0

Local Interpretation:

Occurs(Deposition, ?deposition1)
Occurs(Tilt, ?tilt1)

Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Dcomposition, ?sed-material)
Parameter-of(?tilt1, Theta, ?theta1)

?deposition1.end ≤ ?tilt1.start

Piece-of(?s1, ?sed1)

9. SEDIMENTARY-NO-TILT

S1

S1.orientation@t = 0

Deposition of SED1 (S1 is a piece of SED1)

Scenario Pattern:

Is-Sedimentary(?s1, ?sed-material, ?t)
Orientation-Equal(?s1, ?theta1, ?t)
?theta1 = 0

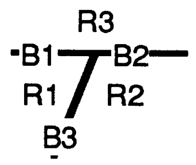
Local Interpretation:

Occurs(Deposition, ?deposition1)

Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Dcomposition, ?sed-material)

Piece-of(?s1, ?sed1)

10. NON-CONFORMABLE-BOUNDARY



$$\begin{tikzcd}
\text{Rock-Creation of ROCK1 (R1 is a piece of ROCK1)} \arrow[dr] & & \text{Rock-Creation of ROCK2 (R2 is a piece of ROCK2)} \arrow[dl] \\
& \text{Rock-Creation of ROCK3 (R3 is a piece of ROCK3)} &
\end{tikzcd}$$

Scenario Pattern:

Faced-Rock-Boundary(?r3, ?b1, ?b2, ?theta, ?r1, ?b3, ?r2, ?t)
Is-Rock-Unit(?r3, ?t)
Above(?r3, ?b1, ?t)
Symmetries: (?r1, ?r2), (?b1, ?b2)

Local Interpretation:

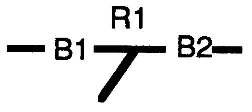
Occurs(Rock-Creation, ?rock-creation1)
Occurs(Rock-Creation, ?rock-creation2)
Occurs(Rock-Creation, ?rock-creation3)

Parameter-of(?rock-creation1, Rock, ?rock1)
Parameter-of(?rock-creation2, Rock, ?rock2)
Parameter-of(?rock-creation3, Rock, ?rock3)
Parameter-of(?rock-creation3, Boundary, ?boundary3)

?rock-creation1.end ≤ ?rock-creation3.start
?rock-creation2.end ≤ ?rock-creation3.start

Piece-of(?r1, ?rock1)
Piece-of(?r2, ?rock2)
Piece-of(?r3, ?rock3)
Piece-of(?b1, ?boundary3)
Piece-of(?b2, ?boundary3)

11. BOUNDARY-CONTINUITY



Boundary-Creation of BOUND1
(B1 and B2 are pieces of BOUND1)

Scenario Pattern:

Colinear-Connected(?b1, ?b2, ?theta, ?t)
Abuts(?r1, ?b1, ?t)
Abuts(?r1, ?b2, ?t)
Is-Rock-Unit(?r1, ?t)
Symmetries: (?b1, ?b2)

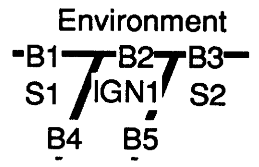
Local Interpretation:

Occurs(Boundary-Creation, ?boundary-creation1)

Parameter-of(?boundary-creation1, Boundary, ?bound1)

Piece-of(?b1, ?bound1)
Piece-of(?b2, ?bound1)

12. INTRUSION-TO-SURFACE



$$\begin{tikzcd}[row sep=large] \text{Deposition of SED1} \\[-2.5em] \text{(S1 and S2 are pieces of SED1)} \arrow[d] \\ \text{Dike-Intrusion of DIKE1 through SED1} \\[-2.5em] \text{(IGN is piece of DIKE1)} \end{tikzcd}$$

Scenario Pattern:

Is-Environment(?env)
Faced-Rock-Boundary(?env, ?b1, ?b2, ?theta, ?s1, ?b4, ?ign1, ?t)
Faced-Rock-Boundary(?env, ?b2, ?b3, ?theta, ?ign1, ?b5 ?s2, ?t)
Above(?env, ?b1, ?t)
Is-Igneous(?ign1, ?ign-material, ?t)
Is-Sedimentary(?s1, ?sed-material, ?t)
Is-Sedimentary(?s2, ?sed-material, ?t)
Symmetries: (?s1, ?s2), (?b1, ?b3), (?b4, ?b5)

Local Interpretation:

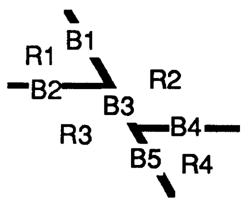
Occurs(Deposition, ?deposition1)
Occurs(Dike-Intrusion, ?dike-intrusion1)

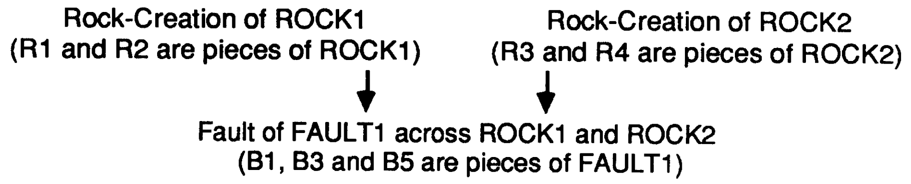
Parameter-of(?deposition1, Rock, ?sed1)
Parameter-of(?deposition1, Upper-Boundary, ?depbound)
Parameter-of(?deposition1, Dcomposition, ?sed-material)
Parameter-of(?dike-intrusion1, Rock, ?dike1)
Parameter-of(?dike-intrusion1, Upper-Boundary, ?intbound)
Parameter-of(?dike-intrusion1, Icomposition, ?ign-material)
Spatially-Intersects(?sed1, ?dike1, ?dike-intrusion1.start)

?deposition1.end $\leq$?dike-intrusion1.start

Piece-of(?ign1, ?dike1)
Piece-of(?b2, ?intbound)
Piece-of(?s1, ?sed1)
Piece-of(?s2, ?sed1)
Piece-of(?b1, ?depbound)
Piece-of(?b3, ?depbound)

13. SIMPLE-FAULT




Rock-Creation of ROCK1
(R1 and R2 are pieces of ROCK1)
Rock-Creation of ROCK2
(R3 and R4 are pieces of ROCK2)
Fault of FAULT1 across ROCK1 and ROCK2
(B1, B3 and B5 are pieces of FAULT1)

Scenario Pattern:

Colinear-Rock-Boundary(?b1, ?b3, ?theta, ?r1, ?b2, ?r3, ?t)
Colinear-Rock-Boundary(?b3, ?b5, ?theta, ?r2, ?b4, ?r4, ?t)
Above(?r1, ?b2, ?t)
Same-Type(?r1, ?r2, ?rock-material1, ?t)
Same-Type(?r3, ?r4, ?rock-material2, ?t)
RL-Side-Of(?r1, ?b1, ?side, ?t)
Symmetries: (?b2, ?b4), (?r1, ?r2), (?r3, ?r4)

Local Interpretation:

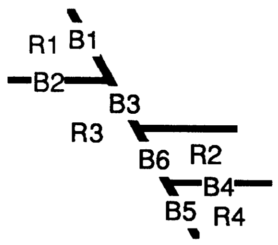
Occurs(Rock-Creation, ?rock-creation1)
Occurs(Rock-Creation, ?rock-creation2)
Occurs(Faulting, ?faulting1)

Parameter-of(?rock-creation1, Rock, ?rock1)
Parameter-of(?rock-creation2, Rock, ?rock2)
Parameter-of(?faulting1, Boundary, ?fault1)
Parameter-of(?faulting1, Ffault-Type, ?ffault-type)
(⇒ (iff (?b1.orientation@?faulting1.start > 90) (?side = 'LEFT))
    (?ffault-type = 'NORMAL-FAULT))
(⇒ (not (iff (?b1.orientation@?faulting1.start > 90) (?side = 'LEFT)))
    (?ffault-type = 'REVERSE-FAULT))
Spatially-Intersects(?rock1, ?fault1, ?faulting1.start)
Spatially-Intersects(?rock2, ?fault1, ?faulting1.start)

?rock-creation1.end ≤ ?faulting1.start
?rock-creation2.end ≤ ?faulting1.start

Piece-of(?r1, ?rock1)
Piece-of(?r2, ?rock1)
Piece-of(?r3, ?rock2)
Piece-of(?r4, ?rock2)
Piece-of(?b1, ?fault1)
Piece-of(?b3, ?fault1)
Piece-of(?b5, ?fault1)

14. EXTENDED-FAULT



$$\begin{tikzcd}
\text{Rock-Creation of ROCK1} \arrow[d] & \text{Rock-Creation of ROCK2} \arrow[dl] \\
\text{Fault of FAULT1 across ROCK1 and ROCK2} &
\end{tikzcd}$$

(R1 and R2 are pieces of ROCK1) (R3 and R4 are pieces of ROCK2) (B1, B3, B5 and B6 are pieces of FAULT1)

Scenario Pattern:

Colinear-Rock-Boundary(?b1, ?b3, ?theta, ?r1, ?b2, ?r3, ?t)
Colinear-Rock-Boundary(?b6, ?b5, ?theta, ?r2, ?b4, ?r4, ?t)
Above(?r1, ?b2, ?t)
Same-Type(?r1, ?r2, ?rock-material1, ?t)
Same-Type(?r3, ?r4, ?rock-material2, ?t)
RL-Side-Of(?r1, ?b1, ?side, ?t)
RL-Side-Of(?r2, ?b6, ?side1, ?t)
?side1 ≠ ?side
Colinear-Connected(?b3, ?b6, ?theta, ?t)
Symmetries: (?b2, ?b4), (?r1, ?r2), (?r3, ?r4)

Local Interpretation:

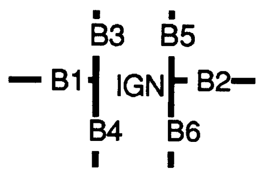
Occurs(Rock-Creation, ?rock-creation1)
Occurs(Rock-Creation, ?rock-creation2)
Occurs(Faulting, ?faulting1)

Parameter-of(?rock-creation1, Rock, ?rock1)
Parameter-of(?rock-creation2, Rock, ?rock2)
Parameter-of(?faulting1, Boundary, ?fault1)
Parameter-of(?faulting1, Ffault-Type, ?ffault-type)
(⇒ (iff (?b1.orientation@?faulting1.start > 90) (?side = 'LEFT))
    (?ffault-type = 'NORMAL-FAULT))
(⇒ (not (iff (?b1.orientation@?faulting1.start > 90) (?side = 'LEFT)))
    (?ffault-type = 'REVERSE-FAULT))
Spatially-Intersects(?rock1, ?fault1, ?faulting1.start)
Spatially-Intersects(?rock2, ?fault1, ?faulting1.start)

?rock-creation1.end ≤ ?faulting1.start
?rock-creation2.end ≤ ?faulting1.start

Piece-of(?r1, ?rock1)
Piece-of(?r2, ?rock1)
Piece-of(?r3, ?rock2)
Piece-of(?r4, ?rock2)
Piece-of(?b1, ?fault1)
Piece-of(?b3, ?fault1)
Piece-of(?b6, ?fault1)
Piece-of(?b5, ?fault1)

15. IGNEOUS-CUTS-BOUNDARY



Boundary-Creation of BOUND1
(B1 and B2 are pieces of BOUND1)

$\downarrow$

Dike-Intrusion of DIKE1 through BOUND1
(IGN is piece of DIKE1)

Scenario Pattern:

```
Is-Igneous(?ign, ?ign-material, ?t)
T-Joint(?ign, ?b3, ?b1, ?b4, ?theta1, ?t)
T-Joint(?ign, ?b5, ?b2, ?b6, ?theta1, ?t)
Colinear(?b1, ?b2, ?theta2, ?t)
```
Symmetries: (?b1, ?b2), (?b3, ?b4), (?b5, ?b6)

Local Interpretation:

```
Occurs(Boundary-Creation, ?boundary-creation1)
Occurs(Dike-Intrusion, ?dike-intrusion1)

Parameter-of(?boundary-creation1, Boundary, ?bound1)
Parameter-of(?dike-intrusion1, Rock, ?dike1)
Parameter-of(?dike-intrusion1, Icomposition, ?ign-material)
Parameter-of(?dike-intrusion1, Boundary, ?intbound)
Spatially-Intersects(?bound1, ?dike1, ?dike-intrusion1.start)

?boundary-creation1.end ≤ ?dike-intrusion1.start

Piece-of(?ign, ?dike1)
Piece-of(?b1, ?bound1)
Piece-of(?b2, ?bound1)
Piece-of(?b3, ?intbound)
Piece-of(?b4, ?intbound)
Piece-of(?b5, ?intbound)
Piece-of(?b6, ?intbound)
```

Appendix B. Causal Models of Geologic Events

This appendix contains descriptions of the object definitions, event model descriptions, definitions of predicates, and domain axioms used in GORDIUS to model "layer-cake" geology.

1. Geologic Objects

Objects are presented as **<Object>** : Parent-Type {<attribute>:<Type>}*. An object inherits all attributes from its parent type, although for completeness all attributes are repeated for each obect.

Point : Temporal-Object
height : Real
lateral : Real

Physical-Feature : Temporal-Object

Environment : Physical-Feature

Air : Environment

Sea : Environment

Geologic-Feature : Physical-Feature
pieces : set of Geologic-Feature
top : Point
bottom : Point
orientation : Angle

Rock-Unit : Geologic-Feature
pieces : set of Rock-Unit
top : Point
bottom : Point
orientation : Angle
thickness : Positive-Real
material : Rock-Material

Up-Thrown-Block : Rock-Unit
pieces : set of Rock-Unit
top : Point
bottom : Point
orientation : Angle
thickness : Positive-Real
material : Rock-Material

```
Down-Thrown-Block : Rock-Unit
    pieces : set of Rock-Unit
    top : Point
    bottom : Point
    orientation : Angle
    thickness : Positive-Real
    material : Rock-Material

Sedimentary : Rock-Unit
    pieces : set of Sedimentary
    top : Point
    bottom : Point
    orientation : Angle
    thickness : Positive-Real
    material : Sedimentary-Rock
    bedding-plane : Gplane

Igneous : Rock-Unit
    pieces : set of Igneous
    top : Point
    bottom : Point
    orientation : Angle
    thickness : Positive-Real
    material : Igneous-Rock

Dike : Igneous
    pieces : set of Igneous
    top : Point
    bottom : Point
    orientation : Angle
    thickness : Positive-Real
    material : Igneous-Rock
    center-plane : Gplane

Batholith : Igneous
    pieces : set of Igneous
    top : Point
    bottom : Point
    orientation : Angle
    thickness : Positive-Real
    material : Igneous-Rock
    bounding-plane : Gplane

Rock-Material : Temporal-Object
    minerals : set of Mineral

Sedimentary-Rock : Rock-Material
    minerals : set of Mineral
    detrial-sediment : Detritus

Shale : Sedimentary-Rock
    minerals : set of Mineral
    detrial-sediment : Mud
```

Sandstone : Sedimentary-Rock
minerals : {quartz}
detrial-sediment : Sand

Conglomerate : Sedimentary-Rock
minerals : set of Mineral
detrial-sediment : Gravel

Igneous-Rock : Rock-Material
minerals : set of Mineral

Mafic-Igneous : Igneous-Rock
minerals : set of Mineral

Granite : Igneous-Rock
minerals : {rock-mineral, quartz, feldspar}

Boundary : Temporal-Object
pieces : set of Boundary
side-1 : set of Physical-Feature
side-2 : set of Physical-Feature

Interior-Boundary : Boundary
pieces : set of Interior-Boundary
side-1 : set of Rock-Unit
side-2 : set of Rock-Unit

Exterior-Boundary : Boundary
pieces : set of Exterior-Boundary
side-1 : {Environment}
side-2 : set of Rock-Unit

Upper-Boundary : Exterior-Boundary
pieces : set of Exterior-Boundary
side-1 : {Environment}
side-2 : set of Rock-Unit

Fault : Boundary
pieces : set of Fault
side-1 : set of Physical-Feature
side-2 : set of Physical-Feature
fault-plane : Gplane
fault-type : one of 'NORMAL, 'REVERSE, 'LATERAL
slip-direction : Angle
slip : Real

Gplane : Abstract-Object
xz-angle : Angle
y-angle : Angle
location : Point

2. Geologic Events

Events are presented using the syntax given in Section 3.2.2 — events have parameter, precondition, effect, and constraint fields. In addition, as with objects, events form a type hierarchy (AKO field), inheriting all attributes from events higher in the type hierarchy.

```
BOUNDARY-CREATION
  Parameters : Boundary : Boundary (created by event)


ROCK-CREATION
  AKO : Boundary-Creation
  Parameters : Boundary : Boundary (created)
               Rock : Rock-Unit (created)
  Effects : Change(=, Boundary.side-1, {Rock}, Rock-Creation)


SUBSIDENCE
  Parameters : Samount : Positive-Real
  Effects :  (For-all ($pt : Point)
                (If Exists-at($pt, Subsidence.start)
                    Change(-, $pt.height, Samount, Subsidence)))


UPLIFT
  Parameters : Uamount : Positive-Real
  Effects :  (For-all ($pt : Point)
                (If Exists-at($pt, Uplift.start)
                    Change(+, $pt.height, Uamount, Uplift)))


TILT
  Parameters : Theta : Angle
  Effects :  (For-all ($gf : Geologic-Feature)
                (If Exists-at($gf, Tilt.start)
                    Change(+, $gf.orientation, Theta, Tilt)))
             (For-all ($pt : Point)
                (If Exists-at($pt, Tilt.start)
                    Change(=, $pt.height, (sin(Theta) * $pt.lateral@Tilt.start +
                                           cos(Theta) * $pt.height@Tilt.start), Tilt)
                    Change(=, $pt.lateral, (cos(Theta) * $pt.lateral@Tilt.start -
                                           sin(Theta) * $pt.height@Tilt.start), Tilt)))
  Constraints : Theta ≠ 0
```

EROSION

```
AKO: Boundary-Creation
Parameters:  Boundary : Boundary (created)
             Elevel : Real
Preconditions: Surface.top.height@Erosion.start > sea-level
Effects:
   Change(=, Boundary.orientation, 0, Erosion)
   Change(=, Boundary.side-1, {Environment}, Erosion)
   Change(=, Boundary.side-2, Ero-Surface(Elevel, Surface, Erosion.start), Erosion)
   (For-all ($ru : rock-unit)
       (If [ Exists-at($ru, Erosion.start) and
             $ru.top.height@Erosion.start ≥ Elevel  and  $ru.bottom.height@Erosion.start < Elevel]
           [ Change(-, $ru.thickness, Erosion-of(Elevel, $ru, Erosion.start), Erosion) and
             Change(=, $ru.top, EroFn1(Elevel, $ru, Erosion.start), Erosion)]))
   (For-all ($bd : boundary)
       (If [ Exists-at($bd, Erosion.start) and
             $bd.top.height@Erosion.start ≥ Elevel  and  $bd.bottom.height@Erosion.start < Elevel]
           Change(=, $bd.top, EroFn1(Elevel, $ru, Erosion.start), Erosion)))
   (For-all ($gf : geologic-feature)
       (If [ Exists-at($gf, Erosion.start) and $gf ≠ Surface and
             Elevel ≤ $gf.bottom.height@Erosion.start]
           Destroyed($gf, Erosion.end)))
   Change(=, Surface.top, EroFn2(Elevel, Surface.top@Erosion.start), Erosion)
   Change(=, Surface.bottom, EroFn2(Elevel, Surface.bottom@Erosion.start), Erosion)
   Change(=, Surface.orientation, EroFn3(Elevel, Surface, Erosion.start), Erosion)
   Change(=, Surface.side-2, Ero-Surface(Elevel, Surface, Erosion.start), Erosion)
Constraints :
   Elevel > sea-level
   Surface.bottom.height@Erosion.end ≤ Surface.bottom.height@Erosion.start
   Surface.top.height@Erosion.end = Elevel
   Surface.bottom.height@Erosion.end ≤ Elevel
   (If Surface.bottom.height@Erosion.start ≥ Elevel
       Surface.bottom.height@Erosion.end = Elevel)
   (For-all ($ru : rock-unit)
       (If [ Exists-at($ru, Erosion.start)  and
             $ru.top.height@Erosion.start ≥ Elevel  and  $ru.bottom.height@Erosion.start < Elevel]
           [ $ru ∈ Ero-Surface(Elevel, Surface, Erosion.start.start) and
             $ru.top.height@Erosion.start.end = Elevel]))
```

EroFn1, EroFn2, Erosion-of, *and* ***Ero-Surface*** *are domain functions, not defined further, that represent different aspects of the erosion event (e.g.,* ***Erosion-of(Elevel, ru, t)*** *represents the amount of* ***ru*** *eroded away; it is constrained to be non-negative and less than the thickness of* ***ru****).*

DEPOSITION

```
AKO : Rock-Creation
Parameters : Boundary : Interior-Boundary (created) ;; The lower depositional boundary
             Rock : Sedimentary (created)
             Upper-Boundary : Upper-Boundary (created) ;; Boundary between rock and sea
             Dlevel : Positive-Real
             Dcomposition : Sedimentary-Rock
Preconditions : Surface.bottom.height < Sea-Level
Effects :
   Change(=, Boundary.side-1, {Rock}, Deposition)
   Change(=, Boundary.side-2, Dep-Bound(Dlevel, Deposition.start), Deposition)
   Change(=, Boundary.orientation, Surface.orientation@Deposition.start, Deposition)
   Change(=, Boundary.top, Dfn4(Dlevel, Surface, Deposition.start), Deposition)
   Change(=, Boundary.bottom, Surface.bottom@Deposition.start, Deposition)
   Change(=, Rock.thickness, Dlevel, Deposition)
   Change(=, Rock.orientation, 0, Deposition)
   Change(=, Rock.material, Dcomposition, Deposition)
   Change(=, Rock.top, Dfn2(Dlevel, Surface, Deposition.start), Deposition)
   Change(=, Rock.bottom, Surface.bottom@Deposition.start, Deposition)
   Change(=, Upper-Boundary.side-1, {Environment}, Deposition)
   Change(=, Upper-Boundary.side-2, {Rock}, Deposition)
   Change(=, Upper-Boundary.orientation, 0, Deposition)
   Change(=, Surface.side-2, {Rock ...}, Deposition)
   Change(=, Surface.orientation, Dfn3(Dlevel, Surface, Deposition.start), Deposition)
   Change(=, Surface.bottom, Dfn1(Dlevel, Surface.bottom@Deposition.start), Deposition)
   Change(=, Surface.top, Dfn1(Dlevel, Surface.top@Deposition.start), Deposition)
Constraints :
   Dlevel ≤ Sea-Level - Surface.bottom.height@Deposition.start
   Surface.bottom.height@Deposition.end = Dlevel + Surface.bottom.height@Deposition.start
   Rock.top.height@Deposition.end = Dlevel + Surface.bottom.height@Deposition.start
   Boundary.top.height@Deposition.end ≤ Surface.top.height@Deposition.start
   Upper-Boundary.top.height@Deposition.end = Dlevel + Surface.bottom.height@Deposition.start
   (For-all ($ru : Rock-Unit)
       (If $ru ∈ Surface.side-2@Deposition.start
           [(If ($ru.bottom.height@Deposition.start ≤
                                Dlevel + Surface.bottom.height@Deposition.start)
               $ru ∈ Boundary.side-2@Deposition.end) and
           (If ($ru.bottom.height@Deposition.start >
                                Dlevel + Surface.bottom.height@Deposition.start)
               $ru ∈ Surface.side-2@Deposition.end)]))
   (If (Surface.top.height@Deposition.start ≤ Dlevel + Surface.bottom.height@Deposition.start)
       Surface.top.height@Deposition.end = Dlevel + Surface.bottom.height@Deposition.start)
   (If (Surface.top.height@Deposition.start = Surface.bottom.height@Deposition.start)
       Boundary.top.height@Deposition.end = Surface.bottom.height@Deposition.start)
```

***Dep-Bound** and **Dfn1–Dfn4** are domain functions, not defined further, that represent different aspects of deposition.*

FAULTING

```
  AKO : Boundary-Creation
  Parameters : Boundary : Fault (created)
               DTB : Down-Thrown-Block (created)
               UTB : Up-Thrown-Block (created)
               Ffault-plane : Gplane
               Ffault-type : One-of 'NORMAL, 'REVERSE
               Fslip : Positive-Real
  Effects :
    Change(=, Boundary.side-1, {DTB ...}, Faulting)
    Change(=, Boundary.side-2, {UTB ...}, Faulting)
    Change(=, Boundary.slip, Fslip, Faulting)
    Change(=, Boundary.fault-type, Ffault-type, Faulting)
    Change(=, Boundary.slip-direction, 90°, Faulting)
    Change(=, Boundary.fault-plane, Ffault-plane, Faulting)
    (For-all ($pt : Point)
        (If Exists-at($pt, Faulting.start) and Is-Point-Of($pt, DTB)
            [ Change(-, $pt.height, Fslip * |sin(Ffault-plane.y-angle)|, Faulting) and
              Change(+, $pt.lateral, Fslip * |cos(Ffault-plane.y-angle)|, Faulting)]))
    Change(=, Surface.orientation,
                    Ffn1(Ffault-plane.y-angle, Surface.orientation@Faulting.start), Faulting)
    (For-all ($gf : Geologic-Feature)
        (If Exists-at($gf, Faulting.start)
            [ Change(=, $gf.bottom, Ffn2(Fslip, Ffault-Plane, $gf, Faulting.start), Faulting) and
              (If Spatially-Intersects($gf, Boundary, Faulting.start)
                (Created ($gf-new1, $gf.type, Faulting.start)
                    (Created ($gf-new2, $gf.type, Faulting.start)
                        Change(=, $gf-new1.material, $gf.material@Faulting.start, Faulting)
                        Change(=, $gf-new2.material, $gf.material@Faulting.start, Faulting)
                        Change(=, $gf-new1.orientation, $gf.orientation@Faulting.start, Faulting)
                        Change(=, $gf-new2.orientation, $gf.orientation@Faulting.start, Faulting)
                        Change(+, $gf.pieces, {$gf-new1, $gf-new2}, Faulting)
                        (If Is-Rock-Unit($gf, Faulting.start)
                            [ $gf-new1 ∈ Boundary.side-1@Faulting.end and
                              $gf-new2 ∈ Boundary.side-2@Faulting.end]))))]))
  Constraints :
    Surface.orientation@Faulting.end ≠ 0
    (For-all ($gf : Geologic-Feature)
        (If Exists-at($gf, Faulting.start)
            Is-Point-Of($gf.bottom@Faulting.end, DTB)))
    (For-all ($bd : Boundary)
        (If Exists-at($bd, Faulting.start) and ($bd.orientation@Faulting.start = 0)
            ($bd.bottom@Faulting.end).height@Faulting.start = $bd.bottom.height@Faulting.start)))
```

Spatially-Intersects*, *Ffn1* and *Ffn2 *are domain functions, not defined further, that represent different aspects of faulting.*

INTRUSION

```
AKO : Rock-Creation
Parameters : Boundary : Interior-Boundary (created)
             Rock : Igneous (created)
             Icomposition : Igneous-Rock
             Iwidth : Positive-Real
Effects :
   Change(=, Boundary.side-1, {Rock}, Intrusion)
   Change(=, Boundary.side-2, Int-Boundary(Rock, Intrusion.start), Intrusion)
   Change(=, Rock.thickness, Iwidth, Intrusion)
   Change(=, Rock.material, Icomposition, Intrusion)
```

BATHOLITHIC-INTRUSION

```
AKO : Intrusion
Parameters : Boundary : Interior-Boundary (created)
             Rock : Batholith (created)
             Icomposition : Igneous-Rock
             Iwidth : Positive-Real
             Ibounding-plane : Gplane
Effects :
   Change(=, Boundary.side-1, {Rock}, Batholithic-Intrusion)
   Change(=, Boundary.side-2,
             Int-Boundary(Rock, Batholithic-Intrusion.start), Batholithic-Intrusion)
   Change(=, Rock.thickness, Iwidth, Batholithic-Intrusion)
   Change(=, Rock.material, Icomposition, Batholithic-Intrusion)
   Change(=, Rock.bounding-plane, Ibounding-plane, Batholithic-Intrusion)
   Change(=, Rock.orientation, 0, Batholithic-Intrusion)
   Change(=, Boundary.orientation, Ibounding-plane.y-angle, Batholithic-Intrusion)
   (For-all ($ru : Rock-Unit)
       (If [ Exists-at($ru, Batholithic-Intrusion.start) and
             Spatially-Intersects($ru, Rock, Batholithic-Intrusion.start)]
           [ Change(-, $ru.thickness, Iwidth * Ifn($ru, Rock), Batholithic-Intrusion) and
             Change(=, $ru.bottom, Ifn1(Ibounding-plane, $ru.bottom@Batholithic-Intrusion.start),
                       Batholithic-Intrusion)]))
Constraints :
   (For-all ($ru : Rock-Unit)
       (If [ Exists-at($ru, Batholithic-Intrusion.start) and
             Spatially-Intersects($ru, Rock, Batholithic-Intrusion.start)]
           [ $ru ∈ Boundary.side-2@Batholithic-Intrusion.end  and
             0 ≤ Ifn($ru, Rock)  and  Ifn($ru, Rock) ≤ 1  and
             $ru.bottom.height@Batholithic-Intrusion.end = Int-Bound-Bottom(Ibounding-Plane)]))
   Boundary.bottom.height@Batholithic-Intrusion.end = Int-Bound-Bottom(Ibounding-Plane)
   Boundary.top.height@Batholithic-Intrusion.end = Int-Top(Ibounding-Plane)
   Rock.top.height@Batholithic-Intrusion.end = Int-Top(Ibounding-Plane)
```

***Int-Boundary, Ifn1, Int-Bound-Bottom** and **Int-Top** are domain functions, not defined further, that represent different aspects of batholithic-intrusion. In addition, the function **Ifn** is defined within the batholithic-intrusion event in terms of constraints on its value.*

DIKE-INTRUSION

```
AKO : Intrusion
Parameters : Boundary : Interior-Boundary (created)
             Rock : Dike (created)
             Upper-Boundary : Exterior-Boundary (created) ;; Boundary with the Environment
             Icomposition : Igneous-Rock
             Iwidth : Positive-Real
             Icenter-plane : Gplane
Effects :
   Change(=, Boundary.side-1, {Rock}, Dike-Intrusion)
   Change(=, Boundary.side-2, Int-Boundary(Rock, Dike-Intrusion.start), Dike-Intrusion)
   Change(=, Rock.thickness, Iwidth, Dike-Intrusion)
   Change(=, Rock.material, Icomposition, Dike-Intrusion)
   Change(=, Rock.center-plane, Icenter-plane, Dike-Intrusion)
   Change(=, Rock.orientation, Icenter-plane.y-angle, Dike-Intrusion)
   Change(=, Boundary.orientation, Icenter-plane.y-angle, Dike-Intrusion)
   Change(=, Boundary.top, Dlfn1(Surface, Dike-Intrusion.start), Dike-Intrusion)
   Change(=, Upper-Boundary.side-2, {Rock}, Dike-Intrusion)
   (If Icenter-plane.y-angle ≠ 0
       Change(=, Upper-Boundary.side-1, {Environment}, Dike-Intrusion))
   (For-all ($gf : Geologic-Feature)
       (If [ Exists-at($gf , Dike-Intrusion.start) and
             Spatially-Intersects($gf , Rock, Dike-Intrusion.start)]
           (Created ($gf-new1, $gf.type, Dike-Intrusion.start)
               (Created ($gf-new2, $gf.type, Dike-Intrusion.start)
                   Change(=, $gf-new1.material,
                               $gf.material@Dike-Intrusion.start, Dike-Intrusion)
                   Change(=, $gf-new2.material,
                               $gf.material@Dike-Intrusion.start, Dike-Intrusion)
                   Change(=, $gf-new1.orientation,
                               $gf.orientation@Dike-Intrusion.start, Dike-Intrusion)
                   Change(=, $gf-new2.orientation,
                               $gf.orientation@Dike-Intrusion.start, Dike-Intrusion)
                   Change(+, $gf.pieces, {$gf-new1, $gf-new2}, Dike-Intrusion)
                   (If Is-Rock-Unit($gf, Dike-Intrusion.start)
                       {$gf-new1, $gf-new2} ⊆ Boundary.side-2@Dike-Intrusion.end)))))
Constraints :
   Boundary.top.height@Dike-Intrusion.end = Rock.top.height@Dike-Intrusion.end
   Boundary.top.height@Dike-Intrusion.end ≤ Surface.top.height@Dike-Intrusion.start
   Boundary.top.height@Dike-Intrusion.end ≥ Surface.bottom.height@Dike-Intrusion.start
```

***Int-Boundary** and **Dlfn1** are domain functions, not defined further, that represent different aspects of dike-intrusion.*

3. Geologic Functions, Predicates and Axioms

This section presents the definitions of functions, predicates, and axioms needed to complete the geologic models given above.

```
On-Surface(?ru, ?t) ≡ (?ru ∈ ?Surface.side-2@?t)

Abuts(?obj, ?bd, ?t) ≡ (?obj ∈ ?bd.side-1@?t) or (?obj ∈ ?bd.side-2@?t)

Orientation-Equal(?obj, ?theta, ?t) ≡
    (?theta - 0.05 ≤ ?obj.orientation@?t) and (?obj.orientation@?t ≤ ?theta + 0.05)

(For-all ($ru : Rock-Unit)
    ($ru.bottom.height@?t < $ru.top.height@?t) and
    Is-Point-Of($ru.bottom@?t, $ru) and
    Is-Point-Of($ru.top@?t, $ru))

(For-all ($bd : Boundary)
    ($bd.bottom.height@?t ≤ $bd.top.height@?t) and
    (If ($bd.orientation@?t = 0)
        $bd.bottom.height@?t = $bd.top.height@?t)))
```

Appendix C. Causal Models for Additional Domains

This appendix contains the domain-specific object definitions, event model descriptions, definitions of predicates, and domain axioms used for the blocks-world and Tower of Hanoi domains. The representations for the semiconductor fabrication domain are found in [Simmons & Mohammed].

1. Blocks-World Objects

```
Supportable-Object : Temporal-Object
    top : set of Supportable-Object
    bottom : set of Supportable-Object

Block : Supportable-Object
    top : set of Block
    bottom : set of Supportable-Object

Table : Supportable-Object
    top : set of Block
    bottom : set of Supportable-Object
```

2. Blocks-World Event Models

```
PUTON-OBJECT
  Parameters:  Source : Block
               Dest : Supportable-Object
  Effects:  (For-all (?b : Supportable-Object)
               (If (Source ∈ ?b.top@Puton-Object.start) and (?b ≠ Dest)
                   Change(-, ?b.top, Source, Puton-Object)))
            Change(+, Dest.top, Source, Puton-Object)
            Change(=, Source.bottom, {Dest}, Puton-Object)

PUTON-TABLE
  AKO: Puton-Object
  Parameters:     Source : Block
                  Dest : Table
  Preconditions:  Clear(Source, Puton-Table.start)
  Effects:     (For-all (?b : Supportable-Object)
                  (If (Source ∈ ?b.top@Puton-Table.start) and (?b ≠ Dest)
                      Change(-, ?b.top, Source, Puton-Table)))
               Change(+, Dest.top, Source, Puton-Table)
               Change(=, Source.bottom, {Dest}, Puton-Table)
```

```
PUTON
  AKO:  Puton-Object
  Parameters:    Source : Block
                 Dest : Block
  Preconditions: Clear(Source,  Puton.start)
                 Clear(Dest,  Puton.start)
  Effects:    (For-all (?b : Supportable-Object)
                 (If (Source ∈ ?b.top@Puton.start) and (?b ≠ Dest)
                     Change(-, ?b.top, Source, Puton)))
              Change(=, Dest.top, {Source}, Puton)
              Change(=, Source.bottom, {Dest}, Puton)
  Constraints: Source ≠ Dest
```

3. Blocks-World Predicates and Domain Axioms

```
Clear(?a, ?t) ≡ (?a.top@?t = {})

On(?a, ?b, ?t) ≡ (?a ∈ ?b.top@?t)

(For-all ($b : Block)
    (For-all ($t : Time)
        ($b ∉ $b.top@$t) and
        ($b ∉ $b.bottom@$t))
```

4. Tower of Hanoi Objects

```
Tower-Object : Temporal-Object
    top : set of Ring
    size : Positive-Real

Ring : Tower-Object
    top : set of Ring
    size : Positive-Real
    rbottom : Tower-Object
    post : Post

Post : Tower-Object
    top : set of Ring
    size : Positive-Real
    rings : set of Ring
    topobj : Tower-Object
```

5. Tower of Hanoi Event Model

```
MOVE-RING
  Parameters:    Ring-Moved : Ring
                 To-Post : Post
  Preconditions: To-Post.topobj.size@Move-Ring.start > Ring-Moved.size@Move-Ring.start
                 Ring-Moved.top@Move-Ring.start = {}
  Effects:
      Change(=, To-Post.topobj, Ring-Moved, Move-Ring)
      Change(+, To-Post.rings, Ring-Moved, Move-Ring)
      Change(=, Ring-Moved.post, To-Post, Move-Ring)
      ;; Don't do the following if trying to move the ring to the post it is already on
      (If (To-Post ≠ Ring-Moved.post@Move-Ring.start)
          [ Change(-, (Ring-Moved.post@Move-Ring.start).rings, Ring-Moved, Move-Ring) and
            Change(=, (Ring-Moved.post@Move-Ring.start).topobj,
                        Ring-Moved.rbottom@Move-Ring.start, Move-Ring)
            Change(=, Ring-Moved.rbottom, To-Post.topobj@Move-Ring.start, Move-Ring) and
            Change(=, (Ring-Moved.rbottom@Move-Ring.start).top, {}, Move-Ring) and
            Change(=, (To-Post.topobj@Move-Ring.start).top, {Ring-Moved}, Move-Ring)])
```

6. Tower of Hanoi Domain Axiom

```
(For-all ($post : Post)
    (For-all ($ring : Ring)
        $post.size@Plan-start > $ring.size@Plan-start))
```

Appendix D. Blocks-World Scenarios

This appendix presents a set of twelve scenarios that completely span that sub-domain of blocks-world problems in which the goal state can contain stacks of up to four blocks high (this includes all the blocks-world problems presented in this report). Even though no individual scenario is applicable to stacks higher than three blocks, this set of scenarios can in fact be composed independently for problems requiring stacks of four blocks.

1. **Scenario Pattern:** *(put A on B; both are clear)*
   Goal Part: On(?a, ?b, Plan-end)
   Initial Part: Clear(?a, Plan-start) Clear(?b, Plan-start)

   Local Interpretation:
   Occurs(Puton, ?puton1)
   Parameter-of(?puton1, Source, ?a) Parameter-of(?puton1, Dest, ?b)

2. **Scenario Pattern:** *(put A on B; O1 is on B; clear B by moving O1 onto object D1; avoid putting O1 on to either A or B)*
   Goal Part: On(?a, ?b, Plan-end)
   Initial Part: Clear(?a, Plan-start) On(?o1, ?b, Plan-start)

   Local Interpretation:
   Occurs(Puton-Object, ?put1)
   Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)

   Occurs(Puton, ?puton1)
   Parameter-of(?puton1, Source, ?a) Parameter-of(?puton1, Dest, ?b)
   (?d1 $\neq$?a) and (?d1 $\neq$?b)
   ?put1.end $\leq$?puton1.start

3. **Scenario Pattern:** *(put A on B; A is not clear)*
   Goal Part: On(?a, ?b, Plan-end)
   Initial Part: On(?o1, ?a, Plan-start) Clear(?b, Plan-start)

   Local Interpretation:
   Occurs(Puton-Object, ?put1)
   Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)

   Occurs(Puton, ?puton1)
   Parameter-of(?puton1, Source, ?a) Parameter-of(?puton1, Dest, ?b)
   (?d1 $\neq$?a) and (?d1 $\neq$?b)
   ?put1.end $\leq$?puton1.start

4. **Scenario Pattern**: *(put A on B; neither are clear)*
Goal Part: On(?a, ?b, Plan-end)
Initial Part: On(?o1, ?a, Plan-start) On(?o2, ?b, Plan-start)

Local Interpretation:
Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)
Occurs(Puton-Object, ?put2)
Parameter-of(?put2, Source, ?o2) Parameter-of(?put2, Dest, ?d2)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?a) Parameter-of(?puton1, Dest, ?b)
(?d1 ≠ ?a) and (?d1 ≠ ?b) (?d2 ≠ ?a) and (?d2 ≠ ?b)
?put1.end ≤ ?puton1.start ?put2.end ≤ ?puton1.start

5. **Scenario Pattern**: *(put A on B, and B on C; all are clear; put B on C before A on B)*
Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)
Initial Part: Clear(?a, Plan-start) Clear(?b, Plan-start)
Clear(?c, Plan-start)

Local Interpretation:
Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
?puton1.end ≤ ?puton2.start

6. **Scenario Pattern**: *(put A on B, and B on C; C is not clear)*
Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)
Initial Part: Clear(?a, Plan-start) Clear(?b, Plan-start)
On(?o1, ?c, Plan-start)

Local Interpretation:
Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 ≠ ?a) and (?d1 ≠ ?b) and (?d1 ≠ ?c)
?put1.end ≤ ?puton1.start ?puton1.end ≤ ?puton2.start

7. **Scenario Pattern**: *(put A on B, and B on C; B is not clear)*

Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)
Initial Part: Clear(?a, Plan-start) On(?o1, ?b, Plan-start)
Clear(?c, Plan-start)

Local Interpretation:
Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 $\neq$?a) and (?d1 $\neq$?b) and (?d1 $\neq$?c)
?put1.end $\leq$?puton1.start ?puton1.end $\leq$?puton2.start

8. **Scenario Pattern**: *(put A on B, and B on C; A is not clear)*

Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)
Initial Part: On(?o1, ?a, Plan-start) Clear(?b, Plan-start)
Clear(?c, Plan-start)

Local Interpretation:
Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 $\neq$?a) and (?d1 $\neq$?b) and (?d1 $\neq$?c)
?put1.end $\leq$?puton2.start ?puton1.end $\leq$?puton2.start

9. **Scenario Pattern**: *(put A on B, and B on C; B and C are not clear)*

Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)
Initial Part: Clear(?a, Plan-start) On(?o1, ?b, Plan-start)
On(?o2, ?c, Plan-start)

Local Interpretation:
Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)
Occurs(Puton-Object, ?put2)
Parameter-of(?put2, Source, ?o2) Parameter-of(?put2, Dest, ?d2)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 $\neq$?a) and (?d1 $\neq$?b) and (?d1 $\neq$?c)
(?d2 $\neq$?a) and (?d2 $\neq$?b) and (?d2 $\neq$?c)
?put1.end $\leq$?puton1.start ?put2.end $\leq$?puton1.start
?puton1.end $\leq$?puton2.start

10. **Scenario Pattern**: *(put A on B, and B on C; A and C are not clear)*

Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)

Initial Part: On(?o1, ?a, Plan-start) Clear(?b, Plan-start)
On(?o2, ?c, Plan-start)

Local Interpretation:

Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)
Occurs(Puton-Object, ?put2)
Parameter-of(?put2, Source, ?o2) Parameter-of(?put2, Dest, ?d2)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 ≠ ?a) and (?d1 ≠ ?b) and (?d1 ≠ ?c)
(?d2 ≠ ?a) and (?d2 ≠ ?b) and (?d2 ≠ ?c)
?put1.end ≤ ?puton2.start ?put2.end ≤ ?puton1.start
?puton1.end ≤ ?puton2.start

11. **Scenario Pattern**: *(put A on B, and B on C; A and B are not clear)*

Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)

Initial Part: On(?o1, ?a, Plan-start) On(?o2, ?b, Plan-start)
Clear(?c, Plan-start)

Local Interpretation:

Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)
Occurs(Puton-Object, ?put2)
Parameter-of(?put2, Source, ?o2) Parameter-of(?put2, Dest, ?d2)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 ≠ ?a) and (?d1 ≠ ?b) and (?d1 ≠ ?c)
(?d2 ≠ ?a) and (?d2 ≠ ?b) and (?d2 ≠ ?c)
?put1.end ≤ ?puton2.start ?put2.end ≤ ?puton1.start
?puton1.end ≤ ?puton2.start

12. **Scenario Pattern:** *(put A on B, and B on C; none of the three blocks are clear)*

Goal Part: On(?a, ?b, Plan-end) On(?b, ?c, Plan-end)

Initial Part: On(?o1, ?a, Plan-start) On(?o2, ?b, Plan-start)
On(?o3, ?c, Plan-start)

Local Interpretation:

Occurs(Puton-Object, ?put1)
Parameter-of(?put1, Source, ?o1) Parameter-of(?put1, Dest, ?d1)
Occurs(Puton-Object, ?put2)
Parameter-of(?put2, Source, ?o2) Parameter-of(?put2, Dest, ?d2)
Occurs(Puton-Object, ?put3)
Parameter-of(?put3, Source, ?o3) Parameter-of(?put3, Dest, ?d3)

Occurs(Puton, ?puton1)
Parameter-of(?puton1, Source, ?b) Parameter-of(?puton1, Dest, ?c)
Occurs(Puton, ?puton2)
Parameter-of(?puton2, Source, ?a) Parameter-of(?puton2, Dest, ?b)
(?d1 ≠ ?a) and (?d1 ≠ ?b) and (?d1 ≠ ?c)
(?d2 ≠ ?a) and (?d2 ≠ ?b) and (?d2 ≠ ?c)
(?d3 ≠ ?a) and (?d3 ≠ ?b) and (?d3 ≠ ?c)
?put1.end ≤ ?puton2.start ?put2.end ≤ ?puton1.start
?put3.end ≤ ?puton1.start
?puton1.end ≤ ?puton2.start