GENERATE, TEST AND DEBUG: COMBINING ASSOCIATIONAL RULES AND CAUSAL MODELS

Reid Simmons and Randall Davis

Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Abstract

We present a problem solving paradigm called generate, test and debug (GTD) that combines associational rules and causal models, producing a system with both the efficiency of rules and the breadth of problem solving power of causal models. The generator uses associational rules to generate plausible hypotheses; the tester uses causal models to test the hypotheses and produce a detailed characterization of the discrepancy in case of failure. The debugger uses the ability to reason about the causal models, along with a body of domain-independent debugging knowledge, to determine how to repair the buggy hypotheses. The GTD paradigm has been implemented and tested in three different domains; we report in detail on its application to our principal domain of geologic interpretation. We also explore in some depth the character of the problems for which GTD is well suited and consider the character of the knowledge required for successful use of the paradigm.

1. Introduction

Problem solving efficiency and broad range of applicability (robustness) are desirable but often incompatible features of AI systems. We present a paradigm called Generate, Test and Debug (GTD) that combines the efficiency of associational rule-based systems with the robustness of reasoning from causal models and we characterize the domains for which GTD may be applicable. An implemented program called GORDIUS uses the GTD paradigm to solve planning and interpretation problems in several different domains — geologic interpretation (the domain for which GORDIUS was initially developed), blocks world planning and the Tower of Hanoi problem.

We have explored the use of GTD primarily for planning and interpretation tasks. Both tasks are of the general form "given an initial state and a final (goal) state, find a sequence of events which could achieve the final state." 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. Examples of planning are block stacking and route planning. Interpretation problems include geologic interpretation and figuring out what happened to the economy.

GTD solves problems in three stages. Figure 1 shows the data and control flow between the generate, test and debug stages.

1. **Generate** — the generator uses associational rules to map from effects to cause. The left-hand side of a rule is a pattern of observable effects and the right-hand side is a sequence of events which could produce those effects. The rules are matched against the final state and the resultant sequences are combined to produce an initial hypothesis — a sequence of events that is hypothesized to achieve the final state.

2. **Test** — the tester simulates the sequence of events to determine the validity of the hypothesis. If the test is successful (i.e., the results of the simulation matches the final state) then the hypothesis is accepted as a solution. Otherwise, the tester produces a causal explanation for why the hypothesis failed to achieve the final state. It then passes this explanation and the buggy hypothesis on to the debugger.

3. **Debug** — the debugger uses the causal explanation from the tester to track down the source of the bugs in the hypothesis. It uses both domain-specific causal models and domain-independent debugging knowledge to suggest modifications which could repair the hypothesis. The modified hypothesis is then submitted to the tester for verification. Alternatively, the debugger has the option to invoke the generator to produce a new hypothesis.

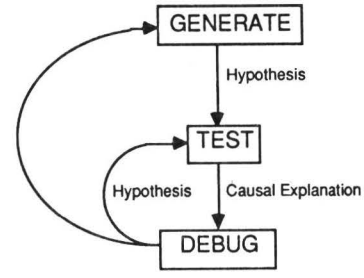

Figure 1. Control Flow of GTD Paradigm

As its name implies, GTD is related to the traditional AI method of *Generate and Test* [Newell]. The debugger is included in GTD to provide more guidance in the search for a solution. In traditional generate and test, the tester reports only a single bit of information — success or failure. In the case of failure, the generator has no indication of what went wrong with the previous hypothesis and so it can only "try again." In GTD, the

debugger uses the test results extensively to focus the search for a valid hypothesis.

The core idea underlying GTD has a long history in AI. Early work by [Sussman] explored "problem solving by debugging almost right plans" as a model of skill acquisition. [Rich & Waters] describe a similar paradigm for the Programmer's Apprentice, calling it AID ("Abstraction, Inspection and Debugging") and more recently, [Hammond] presented a similar method which learned classes of bug repairs for the cooking domain.

Our work advances the state of the art in several ways:

1. Our implementation of the GTD paradigm is relatively domain independent. Just by replacing the associational rules and causal models our program GORDIUS has solved problems in geologic interpretation, all of the blocks-world examples in [Sacerdoti, 77] and the three-ring Tower of Hanoi problem. In addition, the Test and Debug portions of GORDIUS have been used to help diagnose faults in semiconductor manufacturing [Mohammed].

2. Our debugger is fairly robust. We have taken a large step toward developing a general method for debugging sequences of events. The key ideas include reasoning about time and causality using causal models of the domain. We have extended the available methods for reasoning about causal models, since previous domain-independent programs for reasoning about sequences of events (such as [Chapman], [Sacerdoti, 77], [Tate], [Wilkins]) were limited to models which were not complex enough for the geologic interpretation problem. For instance, we need to represent and reason about quantified effects, conditional effects and effects that create or destroy objects.

3. We have analyzed the relationships between the generator and debugger to determine how each contributes to the overall problem solving capabilities of the system. Based on this analysis, we have characterized the domains for which GTD is likely to be useful.

The remainder of this paper describes GTD in more detail and discusses the relative contributions of associational rules and causal models to the GTD paradigm. Section 2 presents the use of GTD in geologic interpretation. Section 3 describes the GTD paradigm in a domain-independent fashion and Section 4 discusses how the associational rules used by the generator and causal models used by the debugger each contribute to the overall performance of the system. Finally, the conclusions characterize domains in which GTD might be applicable.

2. Geologic Interpretation

Our research has focused primarily on the problem known as *geologic interpretation* [Shelton], in which we are given a diagram representing a vertical cross-section of the Earth and a legend indicating the rock types (Figure 2). The task is to infer a sequence of events which plausibly explains how the region came into existence. Figure 3 shows one plausible solution to the cross-section in Figure 2.

We use a simplified model of geology known as "layer cake" geology. Deposition, which occurs when silt in water deposits on the sea bed, creates horizontal sedimentary formations that stack up like the layers of a cake. Erosion occurs when wind abrades exposed rock formations and is assumed to occur horizontally, slicing through the Earth like a knife. Intrusion creates igneous formations when molten rock from below intrudes (pushes) into or through upper rock layers. Faulting splits the Earth and, in our model, moves one side of the fault downwards relative to the other side. Uplift and subsidence move the Earth uniformly up or down, respectively, and tilt rotates the Earth around some origin.

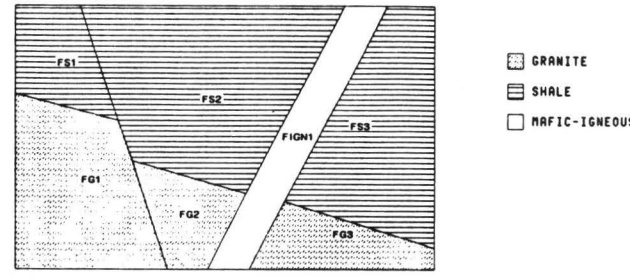

Figure 2. Geologic Interpretation Example

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 3. One Plausible Solution Sequence to Figure 2

2.1 Generation of the Initial Hypothesis

This section describes how GORDIUS interprets the diagram of Figure 2, detailing the knowledge used at each stage of the problem solving process. The first step is to generate an initial hypothesis by matching *scenarios* against the diagram. A scenario is an associational rule that maps from *geologic effects*, which are observable patterns in the diagram, to a *local interpretation*, which is a sequence of events that could have produced the geologic effects.

Geologic effects are represented in terms of topological patterns of edges and faces and their associated geometric constraints. For example, Figure 4 illustrates the "intrusion" scenario, which represents that an igneous formation intruded into an existing rock formation. The pattern in this rule (Figure 4a) matches those parts of the diagram where an igneous rock is between two rocks of the same type and the edges of those rocks are parallel. One match in Figure 2 occurs where **FIGN1** is between **FS2** and **FS3**.

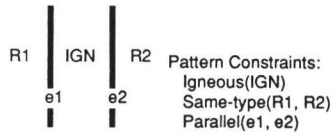

Figure 4a. "Intrusion" Scenario Pattern

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

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

4b. Local Interpretation

The local interpretation of a scenario contains three types of information: 1) the events that occurred to form the observable effects; 2) temporal orderings between events; 3) the objects which are related via a part-whole hierarchy. In the intrusion scenario, the local interpretation (Figure 4b) contains an event that creates rock formation **Rock1** (where **R1** and **R2** are pieces of formation **Rock1**), followed by an intrusive event that creates the igneous formation **IGN**. In addition, the local interpretation includes constraints necessary for testing the hypothesis. For example, the intrusion scenario asserts that the igneous formation is constrained to intersect the rock formation.

Note that scenario patterns do not necessarily imply a unique sequence of events. For example, there are (at least) two plausible explanations for the pattern "sedimentary-over-igneous" where a sedimentary rock is on top of an igneous rock: 1) the igneous rock intruded into the sedimentary rock; 2) the igneous rock intruded into some pre-existing rock, everything was uplifted, the upper layers eroded, exposing the igneous rock, which then subsided, after which the sedimentary rock was deposited on top of the igneous rock. Note that in the first interpretation the igneous rock is younger, while in the second it is older. The generator prefers the first interpretation since it is considered simpler (it is shorter) and hence more likely to have occurred. The second interpretation is used only if the first one leads to an inconsistency.

The local interpretations derived from matching scenarios are combined to yield an initial hypothesis. In this example, four of the fourteen currently defined scenarios were used to produce the initial hypothesis shown in Figure 5. The applicable scenarios are "intrusion" (used twice), "faulting", "erosion" and "sedimentary-over-igneous" (using the first interpretation). Note that the interpretation of Figure 5 differs from the solution in Figure 3 in two important respects. First, the faulting and intrusion are left unordered since the diagram in Figure 2 does not contain enough information to tell which event occurred first. Second, uplift does not appear because, for this example, the uplift event was removed from the local interpretation of the "erosion" scenario; we are using an incomplete scenario in order to provide a simple illustration of the test and debug stages of GTD.

2.2 Testing the Hypothesis

The initial hypothesis is tested using a simulation technique called *imagining* that combines qualitative and quantitative simulations. The imagining technique is described in detail in [Simmons, 83a]; in [Simmons, 83b] we argue the need for both qualitative and quantitative simulations in this domain.

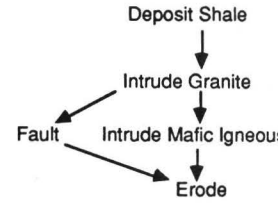

Figure 5. Initial Hypothesis Sequence Generated

The imaginer is given a hypothesis — a partially ordered sequence of events (e.g., Figure 5) — and simulates one of the totally ordered sequences consistent with it. Simulating a total order is necessary because our quantitative simulator can handle only linear sequences of events. Simulating just one of the total orders is sufficient because our task is to produce one plausible interpretation.

The qualitative simulation produces a causal dependency structure describing how the events cause the geologic objects to change over time; the quantitative simulation produces a diagram that is matched against the goal diagram. If the two diagrams match, the hypothesis is considered to be a plausible interpretation. If they do not match, the hypothesis is handed to the debugger for refinement.

In this example, a bug is found during the qualitative simulation: our geologic models state that deposition occurs below sea-level, while erosion occurs above sea-level. Thus, when it comes time to simulate the erosion step in the sequence of Figure 5, the imaginer finds a mismatch between the state of the simulation and the preconditions required to carry out erosion.

2.3 Debugging the Sequence

The debugger attempts to repair the hypothesis by modifying the sequence of events. It uses both the causal dependency structure produced by the tester and geologic domain models to suggest plausible repairs. The modified hypotheses are then tested until a plausible hypothesis is found. Alternatively, if the debugging efforts seem to be moving away from a solution, the generator may be invoked to provide an alternative hypothesis.

The debugger traces back through the causal dependency structure to locate the underlying assumptions made during the generation of the hypothesis that eventually led to the buggy situation. It then uses domain-independent debugging knowledge and the causal geologic models to suggest ways to change the assumptions in order to repair the bug.

Recall that, in our current example, the bug is that the surface of the Earth is below sea-level at the time of the erosion. Two of the many assumptions underlying that bug are 1) some fixed amount of deposition (Dlevel-1) is done and 2) deposition is the last event to affect the height of the Earth's surface. In an attempt to raise the surface of the Earth above sea-level, which would repair

the bug, the debugger first considers increasing Dlevel-1, the amount of deposition. However, the debugger quickly infers that this would not help because our deposition model indicates that deposition can only occur under water, hence no amount of deposition can raise the surface above sea-level.

The debugger next considers inserting processes that can increase the height of the surface. The two possibilities are uplift and tilt. The debugger prefers the uplift process because it infers that inserting a tilt would introduce additional bugs. The debugger thus inserts an uplift event between the deposition and erosion events, adding the constraint that the amount of uplift is enough to raise the surface of the Earth above sea-level (see Figure 6).

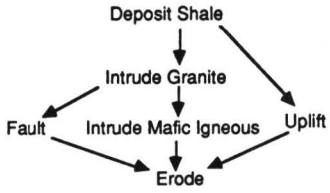

Figure 6. Sequence Generated by GTD Algorithm

This modified hypothesis is then submitted to the tester for verification. As mentioned above, the tester chooses an arbitrary total ordering (see Figure 3) consistent with the partial ordering in Figure 6. This time the simulation completes successfully, producing the diagram in Figure 7. GORDIUS thus concludes that the sequence of Figure 3 is one plausible interpretation of the diagram in Figure 2. This does not imply, however, that all total orderings consistent with the partial order of Figure 6 are plausible solutions.

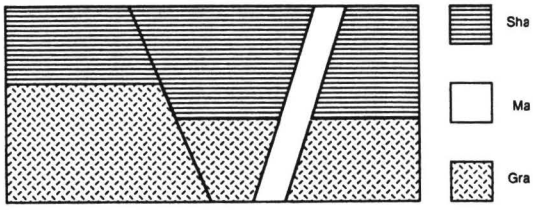

Figure 7. Successful Simulation of Interpretation Example

3. The GTD Algorithm

The previous section illustrated how GTD works in the particular domain of geologic interpretation. This section briefly examines the characteristics of the GTD paradigm as it applies to planning and interpretation problems in general and supplies additional details about the technique.

3.1 Generate

The knowledge base for the generator consists of associational rules that map from effects to cause. In this section we describe the general form of the rules and discuss why the associational rules can be used to generate hypotheses efficiently .

The right-hand side of a rule is a partially ordered sequence of events and associated constraints. In the blocks-world, for example, the rule for interchanging two blocks X and Y where X is initially on Y consists of the event "Puton(X,Z)" followed by "Puton(Y,X)" with the constraint that "Z≠Y," that is, put X somewhere and then put Y on top of X.

The left-hand side of a rule is a pattern consisting of parts that are matched against both the final state and the initial state. The part of the pattern matched against the final state represents observable effects that are produced by the sequence of events given in the rule's right hand side. The part matched against the initial state represents preconditions that must hold for the events to occur. For example, in the block interchange rule described above, "On(Y,X,end)" is matched against the final state and the pattern "On(X,Y,start), Clear(X,start)" is matched against the initial state.*

The inference engine for the generator consists of matching the rule patterns against the initial and final states and combining the sequences of events in a consistent manner. The generator is, in general, quite efficient at producing hypotheses. This arises in part because we use efficient pattern-matching techniques, including discrimination nets.

The main source of the generator's efficiency, however, comes from having associational rules that are nearly independent, that is, the events suggested by applicable rules can be pieced together to form a valid hypothesis, or at worst one that needs only a small amount of debugging.** Thus, an important task in constructing a good rule set is finding the level of granularity where the assumption of independence largely holds. This issue is discussed in more detail in Section 4.

To deal with obvious cases where the assumption of independence fails, the generator possesses a limited form of consistency checking together with the ability to backtrack if an inconsistent hypothesis is detected. The generator can detect inconsistencies in temporal orderings and parameter values For example, it is inconsistent for the sequences "Deposition-1 followed by Intrusion-1" and "Intrusion-1 followed by Deposition-1" to appear in the same hypothesis. Similarly, the generator can infer that one rule suggesting a tilt of 10° is inconsistent with another rule stating that the tilt should be 15°.

Although it is preferable to have the generator do as much consistency checking as possible, testing for global consistency is a relatively expensive procedure that should be done infrequently. Thus, only after the generator produces an initial hypothesis is the resulting sequence of events completely tested.

---

* The scenarios for geologic interpretation differ slightly from this general description. First, they do not refer to the initial state since it is assumed always to be the same — bedrock that is initially below sea-level. Second, for efficiency reasons the scenario patterns are represented as diagrams, however, we are currently re-representing the patterns as sets of predicates in order to use the same generator for all our domains.

** To be precise, the assumption of independence holds only for rules with different patterns since, as noted, the same pattern can be caused by different sequences of events.

3.2 Test

We need to test for two reasons. First, the hypothesis may be incomplete if there is no rule to account for some effect. This is a common problem in developing associational rule sets and occurs when the knowledge engineer overlooks unusual or infrequent situations. Second, combining local interpretations may not yield a globally consistent solution. For example, one of our scenarios matches any single, non-horizontal layer of sedimentary rock and infers that the rock was tilted. If tilting had occurred to that rock, however, all other existing formations would have been tilted as well, which may not be correct.

There are two characteristics the tester must have for GTD to work properly. First, as in the traditional generate and test paradigm, the tester must be correct over the domain of interest relative to the generator. That is, the tester cannot reject valid hypotheses produced by the generator (i.e., allow false negatives) otherwise the GTD algorithm might miss solutions. We use the qualitative simulator of [Simmons, 83a] for our tester since we have found simulation to be a relatively accurate, yet simple, testing technique.

Second, for the debugger to know how to proceed, the tester must return an explanation for why the test failed. The explanation is a causal dependency structure that details how the events in the hypothesis affect the state of the world and explicitly represents when objects *persist*, that is, the intervals of time during which they do not change. These two properties — causality and persistence — provide a foundation for a general method of tracking down and repairing a wide variety of bugs (see Section 3.3).

Figure 8 shows part of the dependency structure produced by our simulator for the bug encountered earlier, in which the surface of the Earth is below sea-level at a time when we would like erosion to occur (the complete dependency structure produced by GORDIUS has almost 200 nodes). It indicates, for instance, that the deposition of shale caused the height of the surface to increase by Dlevel-1 from time t0 to t1 and that that height persisted between t1 and t2.

3.3 Debug

The task of the debugger is to modify the hypothesis in order to repair the bugs found by the tester. The debugger uses the dependency structure created by the tester to track down potential sources of a bug — those assumptions made in generating the hypothesis on which the bug causally depends. This set is generated by collecting the leaf nodes of the dependency structure. In the example of Figure 8, the potential bug sources include 1) nothing changes the height of the surface between time t1 and time t2; 2) time t1 is before time t2; 3) the amount of deposition done is Dlevel-1; 4) sea-level is a constant; 5) all of deposition's remaining (implicit) preconditions hold (this covers all conditions which are beyond the scope of the model, hence are not explicitly represented, such as the fact that deposition can occur only if sediment is present in the water).

We still have the difficult task of deciding which of the potential bug sources is actually to blame and

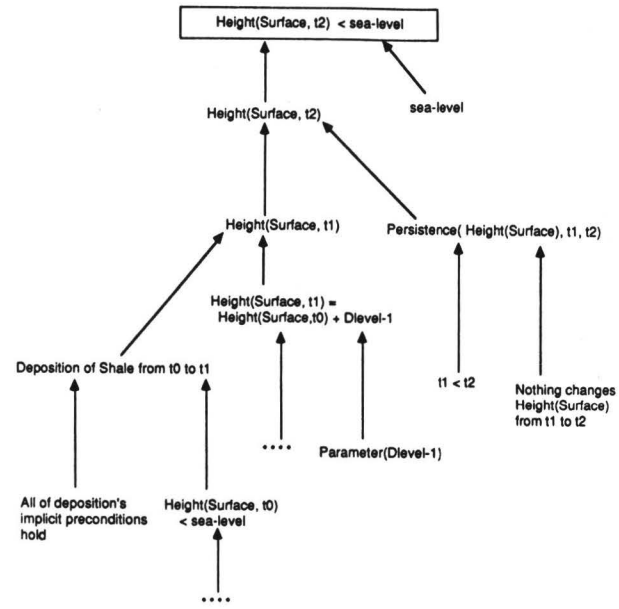

Figure 8. Partial Causal Dependency Structure

determining what to do in order to repair the bug. This is handled by having domain-independent bug repair strategies that suggest modifications to the hypothesis based on an analysis of the domain models, the dependency structure, and the change to the state of the world needed to repair the bug. The bug repair strategies can suggest inserting or deleting an event, replacing an event with a "similar" event, changing parameter values and changing the temporal ordering between events. The debugger chooses among the possible bug repairs using a best-first search strategy whose distance metric is based on the number of unachieved goals and the length of the sequence of events. That is, it prefers short plans that account for as much of the goal state as possible.

Different bug repair strategies are used for different types of assumptions. The current system includes bug repair strategies for: 1) the assumption that nothing changes a value during an interval (i.e., the value persists during that interval); 2) the assumption that a parameter has a particular value; 3) the assumption of a temporal ordering between events; 4) the assumption that all the implicit preconditions of an event hold. Currently the system cannot deal with situations in which the causal models are incorrect, the given initial state is incorrect and something assumed to be constant is in fact varying.

To illustrate the use of repair strategies, consider the assumption "nothing changes Height(Surface) from t1 to t2." Since this is an assumption about persistence, the strategy is to insert a process that can change the value. In this case, we need to change the height of the surface to a value greater than sea-level. In the geologic domain there are several processes that can affect the height of the surface. Two of them, subsidence and faulting, are rejected because their effects are to decrease height. Uplift is suggested because its effect, raising the height of all geologic objects, is what is

needed. Tilting can either increase or decrease height, depending on the direction and origin of the tilt. Thus, the debugger does not know whether adding tilt will actually repair the bug by raising the surface above sea-level. However, the debugger still suggests adding tilt as a possible repair because it considers the resultant, uncertain situation to be an improvement over the currently known, but buggy, situation.

Much of the robustness of the debugger derives from its ability to reason from causal domain models (see Section 4.1). The debugger makes use of the following types of information in determining how to repair bugs: 1) the types of objects which can be changed, created or destroyed by a process; 2) whether a process can affect an object in a particular way, such as changing its height; 3) the magnitude of the effect; 4) the conditions under which the effect occurs; 5) how long the change will last (persistence); 6) whether constraints on parameter values are satisfied.

To facilitate this reasoning, we have developed a representation language for processes that is used in the four domains we have explored (geology, semiconductor fabrication, blocks-world and Tower of Hanoi). Processes are represented as "discrete actions," that is, they specify what the state of the world will be like after the process occurs, but say nothing about what happens while the process is occurring. The language represents time explicitly and describes the state of the world after the process occurs as functions of parameter values and the state of the world before the process occurs. In order to represent complex domains like geology, we have extended the range of traditional STRIPS-like action representations by including conditional effects, universally quantified effects and explicit constraints on parameter values (see [Simmons, 83a]).

4. The Nature of Associational Rules and Causal Models

The utility of GTD is based on the claim that the generator provides an efficient means of synthesizing hypotheses and the debugger provides a robust means of modifying them. In this section, we analyze this claim.

4.1 The Robustness Provided by Causal Models

The claim that the debugger is robust stems in part from the fact that the debugging algorithm described in Section 3.3 uses a fairly general technique to determine how to effect needed changes. However, the robustness of the debugger in a given domain depends largely on the robustness of the causal models it analyzes. Our approach will thus be useful in domains where such causal models are known and where they can be represented in a language that the debugger can analyze. A major research problem is whether such a domain-independent representation language exists, but experience with four rather different domains bolsters our belief that our process representation language and debugger can be used in a wide variety of domains.

A related assumption underlying GTD is that it is easier to construct robust causal models than to construct robust associational rules. Otherwise, it would be less work simply to develop a robust generate and test system. Although 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 working to characterize why and for which domains this assumption holds. One way to do this is to observe that associational rules are typically derived either from experience or from domain models. In any reasonably complex domain one is unlikely, in practice, to experience enough specific cases to span a large fraction of the domain. Thus, to build a robust generator the bulk of the rule set must be derived from domain models. That is, 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.

Reasoning from models is needed in many domains since it is often difficult to ensure that one's rules cover all situations that could arise. For example, in the geology domain there are infrequently occurring classes of events that might easily be overlooked. For example, Figure 4 shows the system's rule about intrusion. Figure 9 shows similar rules, covering infrequently occurring cases in which two or more intrusions coincidentally intrude side by side. These rules are needed to infer that **R1** and **R2** are pieces of the same formation. The large number of possible (although unlikely) interactions among geologic processes makes it likely that a robust generator would need many more special case rules of this type.

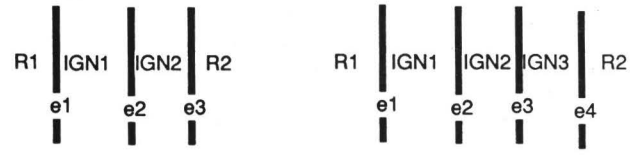

Figure 9. Additional "Intrusion" Scenario Patterns

In some domains, a robust generator might be derived from a formal analysis of the domain. Such was the case with Dendral where group theory was used to prove completeness [Brown & Masinter]. However, in domains such as geologic interpretation it is not clear what types of topological and geometric analyses are needed or even if such an analysis is tractable.

Another possible method for constructing a robust generator would be to simply reindex the causal models (which map from cause to effect) to form associational rules (which map from effects to cause). The problem, as we will discuss below, is that due to its size the resultant rule set would be no more efficient to use than the causal models themselves. Thus, constructing a generator in this way would make it robust but inefficient.

4.2 The Efficiency Provided by Associational Rules

Our experience has shown that the generator is efficient at producing valid or nearly valid hypotheses. Since, as was argued above, most of the associational rules can be derived from the causal models, why is using the using the associational rules so much more efficient

than reasoning directly from causal models?

We believe that this efficiency stems from the fact that the associational rules we use encode two important abstractions of the causal domain models — the encapsulation of interactions and the encoding of problem solving knowledge.

The encapsulation of interactions relates to the assumption, discussed in Section 3.1, that the rules should be nearly independent. A good set of associational rules is one in which each rule covers just enough of the domain so that it can be used nearly independently. For example, the rules in Figure 9 encapsulate the interaction that the formation consisting of pieces **R1** and **R2** was split by the intrusions. If the patterns did not include **R2**, then invalid hypotheses might be produced since the rules would not encode the fact that **R1** and **R2** are created by the same process. On the other hand, if the patterns constrained **R1** and **R2** to be horizontal, the rules would be too specific and valid hypotheses might be missed.

Associational rules also typically encode problem solving knowledge. Such rules derive from an analysis of both the domain models and the problem at hand. For example, the rule that a "fork" is useful in chess combines knowledge about legal chess moves (the domain model) with an analysis of possible moves and countermoves. Rules like these are efficient to use because the generator does not have to repeat the problem solving effort each time the rule is applicable — the problem solving knowledge is "compiled into" the rule. Such rules, however, are specific to the problem for which they were designed. A fork, for instance, is specific to the standard chess game and would be useless if one were allowed two moves at a time or if the object of the game were to lose all one's pieces.

We have described what a good associational rule set is like, but in general it is an open problem how to create such a rule set. It seems that much of the knowledge comes from analyzing how the domain models interact and from practical experience with situations that are likely to occur. This raises the possibility of using the results of debugging an hypothesis to learn associational rules in a manner similar to the work in explanation based learning (e.g., [Mitchell, et al.]). We are currently exploring using GORDIUS' tester and debugger to automatically construct rules in the semiconductor fabrication domain; [Smith, et al.] has explored using causal models to generate geologic rules for doing dipmeter interpretation.

5. Related Work

The core idea in GTD of "debugging almost right plans" was pioneered by [Sussman] in the blocks-world domain. However, Sussman's debugger was not very robust since it used only an ad hoc set of debugging rules rather than reasoning from domain models. [Goldstein] attempted to construct a taxonomy that related bugs to types of errors made during planning. More recently, [Rich & Waters] and [Hammond] have explored paradigms similar to GTD in more complex domains — programming and cooking, respectively. Both systems employ a library of "cliches" to generate hypotheses and both reason from causal models to debug hypotheses. Our work takes a further step in this direction by extending the range of the paradigm — both in terms of its domain-independence and in the complexity of domains which can be handled.

GTD has been used in both planning and interpretation tasks to construct a sequence of events that can achieve a final state from an initial state. The major difference between planning and interpretation is the relative amount of information contained in the final states. In interpretation problems the final state is usually more completely specified since it is a state that has already occurred and is thus, presumably, easier to observe. In planning problems, the initial state is usually more completely specified, especially in comparison with domains such as geologic interpretation where the initial state occurred so long ago that it cannot be specified accurately.

The planning algorithms used by most implementations of domain-independent planners (e.g., [Chapman], [Sacerdoti, 77], [Tate], [Wilkins]) are similar to our debugging algorithm — a subgoal is chosen, a method is found to achieve the subgoal by reasoning from domain models, and the global effect of the action is computed. This is repeated until all the goals are achieved. Our debugger extends this line of research by extending the complexity of models that can be reasoned about. For instance, we represent and reason about quantified effects, conditional effects and effects that create or destroy objects. For example, in geology we need to represent that erosion affects all rocks on the Earth's surface (quantification) — either reducing their thickness or eroding them completely away (conditional effects and destruction of objects).

Such planners, and our debugger, all suffer from the problem of potentially exponential search. This problem led to research aimed at solving problems at different levels of abstraction. Early work by [Sacerdoti, 74] showed the utility of abstraction, but the abstraction technique used — mainly removing preconditions of actions — was fairly simplistic. [Patil] showed that having multiple levels of vocabulary was a powerful problem solving tool. Our work demonstrates the utility of having different representations and inference mechanisms specialized for each level of abstraction (the generator and the debugger).

An interesting point of comparison to GTD is the generate and test system of Dendral [Buchanan]. Buchanan has observed (personal communication) that the test results of Dendral were fairly uninformative — knowing that a peak was the wrong height did little to identify which bonds were wrong since many bonds contributed to each peak. This case illustrates a crucial assumption for GTD — that the goals are not tightly interdependent.

A major open problem with GTD is determining when one is sufficiently "far" from a solution to stop debugging and to ask the generator for a new hypothesis. By running GORDIUS over a wide variety of problems, we hope to detect characteristics of the domain and problem solving strategy that will be useful in determining which stage of the problem solver to use.

6. Conclusions

This paper has presented GTD, a paradigm for combining associational rules and causal models to

achieve efficient and robust problem solving behavior. We have also discussed some aspects of GORDIUS, our implementation of GTD, as it relates to solving problems in geologic interpretation. The following list summarizes many of the domain characteristics that we believe are necessary in order for GTD to be a useful paradigm.

1. Goals are not totally independent. If they were, we would not need both the generator and debugger, since with totally independent goals the debugging approach (solving goals independently) would always lead to a valid solution with little or no search needed. Since, in this case, the generator would lose its advantage of efficiency it would not be worth the extra effort to build both a generator and a debugger.
2. Goals are not totally interdependent. If they were, one would again lose the efficiency advantage of the generator and so would not need both a generator and debugger. With totally interdependent goals the efficiency of the generator becomes as bad as that of the debugger, since the generator approach (using rules independently) would be likely to produce many invalid hypotheses before finding a correct solution.
3. Nearly independent rules can be identified. GTD requires a set of rules whose right-hand sides can be pieced together independently to form a (nearly) valid hypothesis. It might not be possible to find such a set for a given domain, even if the goals are not totally interdependent.
4. Robust causal models must be representable in a language that the debugger can analyze. Although we have developed a representation language suitable for modeling complex domains, it is likely that for other domains our language will need extensions or a different representation altogether will be needed.
5. Experience alone cannot cover most of the domain. If experience can cover the domain then a robust set of associational rules would be easier to construct than robust causal models and the debugger would lose its main advantage of broad range of applicability.
6. The tester must be correct. In particular, it must not allow false negatives.
7. The tester must give causal explanations for any bugs found. Otherwise, the modifications made by the debugger will be random or empirical. However, the tester does not have to use the same models as the debugger if other methods of constructing explanations are available.

The strategy behind GTD is to construct hypotheses using nearly independent associational rules but to include reasoning from causal models as a means of analyzing unforeseen interactions. This research has demonstrated the increased performance and competence exhibited by the GTD paradigm compared with using associational rules or causal models alone.

Acknowledgements

The intellectual history of GTD is long and much research has been done to lay the groundwork for this work, especially that of Sussman's Hacker. We also thank Bruce Buchanan, Dan Carnese, Tom Dean, Ken Forbus, Walter Hamscher, John Mohammed, Chuck Rich, Reid Smith and Peter Szolovits for their help with this research and for comments on this paper.

References

1. Brown, H; Masinter, L. [1974] "An Algorithm for the Construction of the Graphs of Organic Molecules," *Discrete Mathematics*, 8:227.
2. Buchanan, B; Sutherland, G; Feigenbaum, E. [1969] "Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry," in *Machine Intelligence 4,* American Elsevier.
3. Chapman, David [1985] "Planning for Conjunctive Subgoals," MIT AI Technical Report 802.
4. Goldstein, Ira [1975] "Summary of MYCROFT: A System for Understanding Simple Picture Programs," *Artificial Intelligence,* 6:3.
5. Hammond, Kristian [1986] "CHEF: A Model of Case-Based Planning," AAAI-86, Philadelphia, PA.
6. Koton, Phyllis [1985] "Empirical and Model-Based Reasoning in Expert Systems," IJCAI-85, Los Angeles, CA.
7. Mitchell, T; Keller, R; Kedar-Cabelli, S. [1986] "Explanation-Based Generalization: A Unifying Approach," *Machine Learning 1.*
8. Mohammed, John; Simmons, Reid [1986] "Qualitative Simulation of Semiconductor Fabrication," AAAI-86, Philadelphia, PA.
9. Newell, Allen [1973] "Artificial Intelligence and the Concept of Mind," in *Computer Models of Language and Thought,* eds. Schank and Colby.
10. Patil, Ramesh [1981] "Causal Representation of Patient Illness for Electrolyte and Acid-Base Diagnosis," MIT LCS Technical Report 267.
11. Rich, Charles; Waters, Dick [1981] "Abstraction, Inspection and Debugging in Programming," MIT AI Memo 634.
12. Sacerdoti, Earl [1974] "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence,* 5.
13. Sacerdoti, Earl [1977] *A Structure for Plans and Behavior,* American Elsevier.
14. Shelton, John [1966] *Geology Illustrated,* Freeman and Co, chapter 21.
15. Simmons, Reid G. [1983a] "Representing and Reasoning About Changes in Geologic Interpretation," MIT AI Technical Report 749.
16. Simmons, Reid G. [1983b] "The Use of Qualitative and Quantitative Simulations," AAAI-83, Washington, D.C.
17. Smith, R; Winston, H; Mitchell, T; Buchanan, B. [1985] "Representation and Use of Explicit Justifications for Knowledge Refinement," IJCAI-85, Los Angeles, CA.
18. Sussman, Gerald [1977] *A Computer Model of Skill Acquisition,* American Elsevier.
19. Tate, Austin [1977] "Generating Project Networks," IJCAI-77.
20. Wilkins, David [1982] "Domain Independent Planning: Representation and Plan Generation," SRI Technical Note 266.