Northwestern University

The Institute for the Learning Sciences

PROCESS-BASED DIAGNOSIS: AN APPROACH TO UNDERSTANDING NOVEL FAILURES

Technical Report # 48 • December 1993

John William Collins

Established in 1989 with the support of Andersen Consulting

PROCESS-BASED DIAGNOSIS: AN APPROACH TO UNDERSTANDING NOVEL FAILURES

John William Collins

December 1993

The Institute for the Learning Sciences
Northwestern University
Evanston, IL 60201

This research was supported by the NASA Langley Research Center. The Institute for the Learning Sciences was established in 1989 with the support of Andersen Consulting. The Institute receives additional support from Ameritech and North West Water, Institute Partners, and IBM.

PROCESS-BASED DIAGNOSIS: AN APPROACH TO UNDERSTANDING NOVEL FAILURES

BY

JOHN WILLIAM COLLINS

B.S., Rose-Hulman Institute of Technology, 1979
M.S., University of Illinois, 1988

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

Process-Based Diagnosis: An Approach to Understanding Novel Failures

John William Collins, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1994
Kenneth D. Forbus, Advisor

This thesis describes a diagnostic technique for explaining unanticipated modes of failure in continuous-variable systems. Previous approaches in model-based diagnosis have traditionally suffered from either a dependence on explicit fault models or a tendency to produce unintuitive results. This research aims at achieving the explanatory power of explicit fault models, without sacrificing the robustness of consistency-based diagnosis. The unique compositional nature of the process-centered models of Qualitative Process Theory makes the application of model-based diagnostic techniques both non-trivial and rewarding. Rather than relying on explicit fault models, this approach utilizes a general *domain theory* to model the broken device. Given a sufficiently broad domain theory, symptoms are explained in terms of a transformed physical structure. *Generative* fault models replace explicit, pre-enumerated fault models, thereby increasing robustness for identifying novel faults. This approach combines the efficiency of the consistency-based approach with the explanatory power of abductive backchaining. Candidates generated using a consistency-based approach are used to focus the abductive search for a structural model of the failed system. An implementation built on a modified ATMS and an incremental qualitative envisioner is tested on a number of examples. The systems examined are taken primarily from the domain of thermodynamics, but also include some simple circuits.

To Brian

ACKNOWLEDGMENTS

I extend special gratitude to the following colleagues and friends:

Ken Forbus, for allowing me to be a part of the Qualitative Reasoning Group; for being there when I needed help; and for giving me the freedom to explore on my own. Thanks also for developing QP theory, the cornerstone of this research.

Dennis DeCoste, for our two years of collaboration, without which this work would not be possible. I benefitted greatly from your unique perspective and your ability to get to the heart of every issue. Our countless hours of discussion evolved into a vocabulary and language which (at best) only we two could understand. I will miss that high bandwidth channel.

Gordon Skorstad, for many enlightening discussions about everything under the (rising) sun, from conspiracy theories to the latest Macintosh benchmarks, and occasionally, even research.

Scott Bennett, for answering my countless UNIX questions and helping me keep the RTs alive; thanks also for showing me that success comes from believing, trying and doing.

Dan Oblinger, for getting me started with abductive backchaining, and for tolerating my incessant questions.

My committee members: Seth Hutchinson, Jean Ponce, David Wilkins, and Marianne Winslett, for agreeing to serve on my committee, and for all of your helpful suggestions.

Johan deKleer, Raymond Reiter, Randall Davis and countless other researchers in Model-Based Diagnosis, for assembling such a solid foundation upon which to build.

Jane Malin (JSC), Kathy Abbott (LRC) and the National Aeronautics and Space Administration, for providing the funding for this research.

Finally, my wife Bea, for supporting me and sticking with me through it all, even when two years turned into seven.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

"Diagnosis begins with a failure to understand."

1.1 Introduction

We live in a manufactured world. Every day of our lives, we are surrounded by machines, appliances, vehicles, equipment and other devices intended to increase our productivity, our mobility, and the quality of our lives. In addition to death and taxes, there is one inescapable truth in this synthetic world: machines break down. Whether it is due to planned obsolescence, misuse, poor design or unavoidable wear and fatigue, nearly every machine will require repair sometime during its lifespan. Before a machine may be repaired, it is necessary to understand the nature of the failure. The general act of understanding the nature, location and extent of failure in a malfunctioning machine is called *diagnosis.*

As machines become more complex, there are more ways for them to fail, and the failures become more difficult to understand. Thus it becomes increasingly important to divert a portion of our overall technological effort toward automating the diagnosis of our creations. In the past two decades, automated diagnosis has been an active area of research within the Artificial Intelligence community. There have been many successes, but much still remains to be done.

This research investigates the application of process-centered qualitative models to the problem of understanding unanticipated modes of failure. The model-based approach to diagnosis is extended by applying a general domain theory to understanding a device after failure. By viewing a failure as residing in the model of the device rather than the device itself, and recognizing that the device continues to obey the laws of nature, it is possible to apply the same knowledge to derive expectations and to determine why they fail to be met. The approach taken in this research results in a novel combination of consistency-based and abductive approaches to model-based diagnosis, yielding greater power and effeciency than either approach alone.

1.2 The Nature of Diagnosis

Diagnosis is the act of identifying a set of *faults* given a set of *symptoms.* A fault consists of a deviation of the physical structure of the device relative to its form prior to the failure. A symptom is an observed deviation in the behavior of the device relative to the desired or expected behavior, which presumably would have been observed had the failure not occurred.

Defined in these terms, diagnosis is the act of inferring the structural deviations of a failed device which explain an observed deviation from expected behavior.

1.2.1 An Example

Consider the following situation. A technician in a chemical plant is monitoring a vat of sulfuric acid, and notices that the level of acid in the vat is slowly dropping. This observation is unexpected because all valves connected to the vat register as closed. The technician is concerned, knowing that the acid must be going somewhere. Three possibilities immediately come to mind, all of them potentially catastrophic.

One possible explanation is that the acid is boiling away. This would require the acid to be at its boiling point, and would result in an observable (and deadly) cloud of sulfuric acid vapor above the vat. Being still alive, the technician quickly rules out this possibility.

A second possibility is that one of the valves is not really closed, but is giving a false indication due to a sensor failure. This possibility deserves further attention; an unexpected flow of sulfuric acid could be catastrophic to a downstream process. Flow meters on the connected pipes all register zero flow, so this hypothesis appears unlikely. It would require three separate failures: failure of the valve to close on command; failure of the valve position indicator;[1] and failure of the associated flow meter.

The only remaining possibility is that the vat has sprung a leak. Such a failure could threaten the lives of the technician and other plant employees, and so must be taken very seriously. A leak would not register on any flow meter, but would leave an observable accumulation of acid somewhere beneath the vat. After spotting a pool of liquid just under the vat, the technician quickly notifies emergency crews and initiates corrective actions to empty the vat.

This example illustrates several important aspects of diagnosis. First, note that the technician applies commonsense knowledge to understanding the failure. For example, knowing that "the acid must be going somewhere" is a form of the law of conservation of matter, and is common knowledge.

When expert knowledge is required, it may still be of a general nature. Identifying the set of candidate explanations for the example above requires some general knowledge of fluids, and some expertise may be required to be confident that no other possibilities exist. Still, this knowledge is very general, and is not limited to vats of sulfuric acid in a chemical plant.

The search for candidate explanations is focused by knowledge of the domain and the nature of the symptoms which originally led to a suspicion of failure. In the example above, the observation that the level of acid is dropping when it is expected to be constant triggers the technician to search for possible causes of a falling fluid level. Two processes come to mind: flow and boiling.

Candidate explanations are evaluated by simulating their causes and effects. When considering a particular candidate explanation, the technician mentally simulates the effects of a hypothesized failure, looking for observable or testable behaviors or states. For example, boiling is ruled out because no gaseous sulfuric acid is visible above the vat. The technician could also look for necessary prerequisites for an hypothesized failure; boiling could be ruled out by observing that the acid is not hot enough to boil. Prerequisites and consequences of an hypothesized failure provide validation criteria by which hypotheses are refuted or confirmed.

[1]These two failures might not be independent; failure of the valve position indicator could explain the failure of the valve to close.

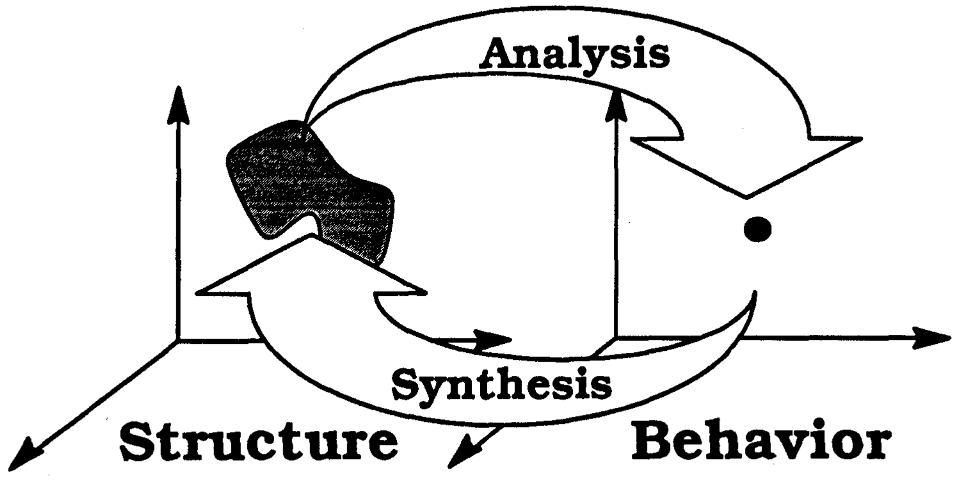


Figure 1.1: Diagnosis is a Synthesis Problem

Candidates may involve unknown objects and processes. The technician knows that flows require a path and a destination, and so proceeds to evaluate possible flows through known paths. Indications of closed valves and zero flow rates cause these candidates to appear unlikely. Finally the technician hypothesizes an unknown path and destination—a leak.

1.2.2 Diagnosis as Synthesis

Within the analysis/synthesis dichotomy, diagnosis is best viewed as a form of synthesis. Whereas analysis involves a mapping from structure to behavior, synthesis involves the inverse mapping from behavior to underlying structure (see Figure 1.1). Other examples of synthesis include design, planning, scientific discovery and understanding observations.

Synthesis is generally harder than analysis, for two reasons. First, the mapping from behavior to structure is not always unique. There may be several different physical devices whose outward behaviors are indistinguishable. This problem can be aggravated by the lack of a complete behavioral description. Second, the mapping from behavior to structure is generally not well understood. Synthesis usually requires a verification phase, where a candidate structure is analyzed to verify that it produces the desired behavior.

Diagnosis differs from other synthesis problems in that, typically, the physical structure of the device prior to failure is already well understood. This knowledge can be of immense help in understanding the device after failure. Diagnosis seeks to explain why a device suddenly or gradually ceases to perform its intended function. Given that the laws of nature and the functional requirements of the device have remained fixed, it follows that the physical structure of the device has changed in some way.

1.2.3 Failure: Device vs. Model

Expectations for the behavior of a device may be based on experience with the same or a similar device, or on fundamental first-principles knowledge of the laws of nature. In the latter case, knowledge of the physical structure of the device can be used to simulate the device in operation, leading to predictions of observable behavior. Such knowledge defines a *model* for the device, and is the essential ingredient in model-based diagnosis research.

When the behavior of a device deviates from that predicted by our model of it, the problem is traditionally viewed as residing within the device. Model-based diagnosis takes the opposite view that the problem lies within the model from which our expectations are derived. Specifically, the model has failed to keep up with changes in the device. Regardless of the severity of the "damage" to the device, it continues to operate "correctly", viewed from the perspective of the laws of nature. It is our expectations which need adjusting.

By recognizing that a broken or damaged device obeys the same laws as a fully functional one, we can apply the same knowledge to understanding failure as we apply to understanding behavior in general. Specifically, the knowledge used to produce our original expectations can be used again to explain why they were not met.

1.3 Overview of Process-Based Diagnosis

The approach to diagnosis presented in this thesis attempts to achieve a deeper understanding of a failed device, through the application of first-principles qualitative models. We claim the following:

Failures reside within the model, not the device. A broken device continues to obey the natural laws; it is our model of it which has failed to keep pace with changes within the device. Diagnosis is an attempt to understand and repair a failed model.

Diagnosis should explain failures. For an automated diagnosis system to be considered intelligent, it must go beyond localizing a failure to one or more components; it must seek a deeper understanding of the structure of the failed device. Any hypothesized structure must be consistent with observed symptoms, and ideally, should predict them.

Many failures result in a modified process structure. A broken device may be understood in terms of its underlying active processes. A fault may activate new processes or deactivate existing ones. For example, a leak in a fluid system may be seen as an instance of a fluid flow process. No special fault model for leaks is required, so long as fluid flows are already part of the domain knowledge.

Some kinds of knowledge are beyond suspicion. The process-centered models used in this research are composed from a general domain theory and a specific structural description for a device. The domain theory is assumed to be correct, and remains applicable to the broken device. Diagnosis is a search for a modified structural description of the device after failure.

Process structure is more yielding than physical structure. By initially focusing on the active process structure rather than the underlying physical structure of a failed device, the search space is greatly reduced. The search is further focused by knowledge of which quantities are changing unexpectedly, given that each quantity can only be influenced by a few types of processes.

Structure added by a failure violates closed-world assumptions. Explicit representation of closed-world assumptions allows us to question the belief that what we know about is all there is. If a failure introduces new structure, it may be initially detected as a closed-world assumption violation.

This research makes the following contributions:

A unique combination of two diagnostic approaches.
This research combines consistency-based diagnosis with abductive backchaining. The consistency-based approach is used initially to search for minimal perturbations to the original model which are consistent with observed symptoms. This provides the focus and starting point for an abductive search for any missing fragments of the model. By combining the strengths of the two approaches, we effectively divide up the problem so as to increase both inferential power and reasoning efficiency.

An efficient minimal retraction algorithm. Like many problems in default reasoning, diagnosis involves finding minimal retractions to a set of default beliefs to resolve an inconsistency. Based on Reiter's *DIAGNOSE* algorithm [68], the minimal retraction algorithm developed in this research avoids much of the redundancy of the original solution.

A General ATMS-based Abduction Algorithm. Diagnosis is a form of abduction, as it involves a search for explanations of observed phenomena. This research contributes an ATMS-based abduction algorithm, called `GA3`, which is used to postulate missing structure which could explain a closed-world assumption violation.

The Process-based Diagnostic Engine. `PDE` is the implementation of the approach and algorithms outlined above, based on an incremental qualitative envisioner (`IQE`) and a hybrid ATMS (`CATMS`). The performance of `PDE` is demonstrated on a number of examples.

1.4 Organization of the Thesis

The remainder of the thesis is organized as follows:

Chapter 2 provides an overview of process-based diagnosis. The chapter begins with a brief review of qualitative reasoning and model-based diagnosis, explaining the context in which this research evolved. The chapter then describes process-based diagnosis, emphasizing its unique contributions relative to previous model-based approaches. The chapter concludes with a high-level overview of the implementation of this approach, the Process-based Diagnostic Engine (`PDE`).

Chapter 3 describes the modeling issues addressed by this research. The modeling language and the rule system supporting the model are presented. The chapter examines the rules required to capture the domain-independent constraints of qualitative reasoning, and emphasizes the benefits of a uniform, inspectable representation for all aspects of the model. Chapter 3 also includes a description of the types of defeasible assumptions considered in this research, including the closed-world assumption and its relation to default reasoning.

Chapter 4 discusses the generation of candidate diagnoses, using an efficient algorithm for finding minimal hitting sets of the set of conflicts. These represent the minimal retractions of default beliefs which eliminate an inconsistency. This approach builds on the earlier work of Reiter [68] and deKleer [26], by improving the efficiency of an earlier algorithm and extending it beyond its original intended scope.

Chapter 5 presents an algorithm for abductive backchaining and its application to verifying diagnostic candidates. This algorithm, based on an assumption-based truth maintenance system, is an extension of an earlier approach [57], overcoming limitations which precluded its direct application to this research.

Chapter 6 illustrates process-based diagnosis on a number of examples drawn primarily from the domain of thermodynamics. These examples serve to elucidate the process-based approach to diagnosis, substantiate the claims of the thesis, and demonstrate the performance of the implementation of the approach, PDE.

Finally, Chapter 7 summarizes the major claims and contributions of the thesis, discusses some of the limitations of the approach and of the implementation, and identifies areas for future research.

Chapter 2

Process-Based Diagnosis

2.1 Introduction

This chapter provides an overview of *Process-Based Diagnosis*, an idea conceived in the union of Qualitative Process (QP) Theory and model-based diagnosis. However, process-based diagnosis is more than a simple application of model-based diagnostic techniques to the process-centered models of QP theory. The unique compositional nature of the process-centered models of QP theory makes the marriage of these two ideas particularly fertile. This chapter describes the elements of this synthesis, and shows how they combine to yield an efficient and robust approach to diagnosis.

This chapter is organized into two parts, each consisting of two sections. The first part briefly reviews the elements from which process-based diagnosis was conceived. Section 2.2 highlights the basic ideas underlying qualitative reasoning and in particular, Qualitative Process theory. Section 2.3 summarizes model-based diagnosis, and briefly reviews the approaches taken by other researchers in this area. In addition to defining the context in which these ideas evolved, this section provides an appreciation of the problem to be solved, and shows the limitations of previous solutions.

The second part of this chapter provides a concise description of the key ideas of process-based diagnosis, which are expanded upon in the rest of this thesis. Section 2.4 describes the theory behind process-based diagnosis, including the hurdles and rewards of applying model-based diagnosis techniques to the process-centered qualitative models of Qualitative Process Theory. Section 2.5 describes the *Process-based Diagnostic Engine* (PDE), the implementation of the process-based diagnostic approach. The chapter concludes with a summary of the major claims underlying process-based diagnosis.

2.2 Overview of Qualitative Reasoning

This research investigates the use of process-centered qualitative models in diagnosing unanticipated failures in continuous-variable systems. This section provides the relevant background material on qualitative reasoning and in particular Qualitative Process Theory, which is the basis for the models used in this research.

Qualitative reasoning attempts to describe or predict the continuous behavior of physical devices, without manipulating actual numbers. Typically this is accomplished by partitioning the range of values of a quantity into a small number of interesting subregions. A common example of such a partitioning is the sign operator, which maps a quantity into three regions: *positive, negative* and *zero.* A generalization of this approach allows comparisons between any two quantities (instead of just comparisons with zero). The leverage of qualitative reasoning comes from making only those distinctions which are relevant to the problem at hand.

Unlike numerical simulation, qualitative reasoning provides the ability to reason from incomplete information, and with potentially fewer computations. In addition, qualitative reasoning provides a psychologically plausible account for much of human reasoning about certain kinds of physical systems. People clearly do not compute exact solutions or run numerical simulations in predicting or understanding an outcome. Instead, we often describe behaviors in qualitative terms, without reference to magnitudes or units. Yet we are able to predict behaviors and postulate the underlying mechanisms responsible for an observation. This ability to draw conclusions with minimal amounts of data suggests that qualitative reasoning is well suited for knowledge intensive problems like diagnosis.

2.2.1 Qualitative Process Theory

Qualitative Process (QP) theory [37] was developed as a model of commonsense reasoning about continuous change in systems of physical components. QP theory is based on the tenet that all change results from the action of *processes.* An active process influences quantities, causing them to change over time. The possible types of processes are defined in terms of the conditions under which they are active and the effects on the behavior of the modeled system.

A second tenet of QP theory is that qualitatively interesting changes in behavior coincide with changes in certain inequalities. For example, the flow between two containers stops exactly when their levels become equal. Thus the qualitative state of the modeled system may be described in terms of a collection of inequality relations. By considering how these relations can change over time, it is possible to simulate the sequence of qualitative behaviors of the system.

QP theory prescribes a language for expressing general theories for classes of physical entities and phenomena within a particular domain. QP theory also suggests procedures for automatically deriving a causal model for a given scenario within a domain. A *model* in QP theory is composed from two separate constituents: a *domain theory* defining the causal relations underlying a general domain (such as thermodynamics); and a *scenario description* of the specific system being modeled, indicating its components and their connectivity. QP theory offers *composability,* in that an unbounded number of distinct scenario descriptions can be modeled using a single domain theory.

Quantities, Numbers and Inequalities Quantities play a central role in QP theory. A *quantity* represents some continuous parameter which is relevant to the model. Associated with each quantity are two *numbers*: its amount and derivative (w.r.t. time). An *inequality* relates two numbers by defining the ordering between them. Because quantities can have non-zero derivatives, inequality relations can change over time. Inequalities define the qualitative state of the modeled system.

Objects, Attributes and Relations Objects are defined in QP theory as belonging to one or more *entity classes.* Each entity class has a set of attributes which are inherited by its instances. Typical attributes include the quantities of an object, the causal and inequality relations between them, and more general entity classes to which the object belongs. Connectivity relations between two or more objects define the structural connections of the scenario.

Views and Processes Views and processes are conditional descriptions of state and behavior, respectively. An instance of a view or process exists for each combination of objects matching the *individuals* of the corresponding *view type* or *process type*, as defined by the domain theory. An existing view or process instance becomes *active* when its *preconditions* and *quantity conditions* are satisfied. Preconditions represent modeling assumptions or controls which are manipulated by some external agent (such as a manually-operated valve). Quantity conditions are inequality relations required to hold for the activation of the view or process. As a consequence of the active instance, the *relations* of the view or process become true.

Causal Influences Processes differ from views in that processes have *influences.* An active process instance directly influences one or more quantities; the degree of influence is equal to the *rate* of the process. For example, a liquid flow process directly influences the amounts of liquid at the source and destination of the flow. It is possible for more than one process to directly influence the same quantity; the derivative of the influenced quantity is equal to the sum of the rates of the influencing processes. Processes and their influences are viewed as the causal origin of all change; if there are no active processes, then all quantities must be constant.

Quantities can be indirectly influenced by a process, through chains of *qualitative proportionalities.* As with direct influences, qualitative proportionalities may be either positive or negative. The relation: $Q_1 \propto_{Q+} Q_2$ states that Q_1 will change in the same direction as Q_2 when all other determiners of Q_1 are constant. In other words, Q_1 is monotonically increasing in Q_2. Likewise, $Q_1 \propto_{Q-} Q_2$ has Q_1 monotonically decreasing in Q_2. In both cases it is implicitly assumed that a change in Q_2 *causes* a change in Q_1. Therefore the qualitative proportionalities must run outward from directly influenced to indirectly influenced quantities. Since these relations are viewed as imputing causality, loops among qualitative proportionalities are not allowed.

Qualitative Modeling of Thermodynamics The modeling language developed under QP theory has allowed the accumulation of a significant domain theory for thermodynamics [7]. This domain theory provides a rich testbed for the approach to diagnosis presented in this thesis. Chapter 3 describes the modeling language in more detail, while Chapter 6 includes several examples of thermodynamic systems modeled using this domain theory. The domain theory itself is included as Appendix A.

2.3 Overview of Model-Based Diagnosis

Diagnosis is a knowledge-intensive cognitive task, and has been an area of active research in AI for more than two decades. Early efforts aimed at automating diagnosis were based on direct association of symptoms with underlying disorders, acquired from an expert diagnostician. Expert systems such as MYCIN [73] showed impressive diagnostic abilities within their intended domain—often performing as well or better than their human counterparts.

Because symptom-disorder associations lack hierarchical structure, they are sometimes referred to as *shallow* knowledge. Due to the the relatively shallow nature of their rule base, early expert systems did not generalize well beyond their original intended domain [2]; they tended to be *brittle* when applied to novel situations or failure modes. Associational rules are generally easier to construct for a given device than a more general theory for its domain; but there is less amortization of the cost of model construction, and less transfer to new devices. Still, for complex systems in poorly understood domains, associational rules may be the only option.

In an attempt to achieve greater generality, some researchers ([22, 13, 46]) began to investigate a model-based approach to diagnosis. In certain well-understood domains such as electronics, models of individual components and their failure modes were developed and used to automatically compose system-level models from schematic-level descriptions of circuits. Qualitative models have been shown to be sufficiently rich for much of this style of reasoning. Model-based diagnosis can reason effectively with novel assemblies composed from the known component types. However, these early approaches still suffered from the need for explicit fault models. That is, they were still brittle when applied to unanticipated failure modes.

2.3.1 Overview of Consistency-Based Diagnosis

In the late 1970's and early 1980's, researchers [13, 22] developed what is commonly known as the consistency-based approach to diagnosis. A technique called *constraint suspension* [13] was introduced to overcome the dependency on explicit fault models, while still identifying components whose failure could account for observed symptoms. If suspending the constraints associated with a suspected component eliminates the conflict between observations and expectations, then that component may be responsible for the observed symptoms. Constraint suspension does not require explicit fault models, and extends to modes of failure which were not anticipated by the model builder.

GDE [26] impliments consistency-based diagnosis using an assumption-based truth maintenance system (ATMS), and handles multiple faults in a natural way. GDE utilizes ATMS label propagation to identify the components involved in conflicts between actual and predicted behaviors. Minimal sets of components which intersect with every conflict set are identified as candidate diagnoses. This approach to generating candidates is discussed in greater detail in Chapter 4.

Consistency-based diagnosis works well in domains such as digital electronics, where signals propagate only in one direction and interactions are minimal. It is of little help in circuits involving feedback, for example, where the failure of any one component could potentially account for any anomalous behavior. Another limitation of consistency-based diagnosis is that symptoms are not actually *explained.* Without explicit fault models, consistency-based diagnosis is unable to verify that some physically-realizable mode of failure in the suspected components could actually account for the observed behavior.

Because consistency-based diagnosis places no restrictions on the behavior of a failed component, it is sometimes unable to rule out intuitively absurd candidates. Figure 2.1 depicts an example first used by Struss and Dressler [75] to illustrate the shortcomings of the consistency-based approach. Three light bulbs are connected to a battery in parallel, and only one of the bulbs is illuminated. In addition to the intuitive explanation that two of the bulbs have failed, consistency-based diagnosis finds a candidate claiming that the battery and the one illuminated bulb have failed (i.e., the battery has no voltage and the bulb is glowing when it should not). Common sense tells us that light bulbs cannot glow without an applied voltage, no matter how they fail; but the traditional consistency-based approach knows no such thing.

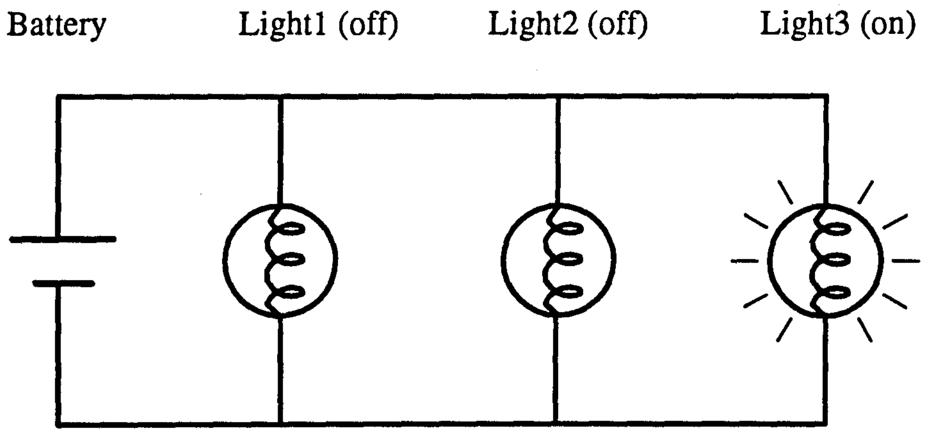


Figure 2.1: A Battery and Three Light Bulbs in Parallel

2.4 The Theory Behind Process-Based Diagnosis

This thesis is based on the following observation: Regardless of the nature or extent of a failure, a broken device continues to behave according to the laws of nature. Thus it should be possible to understand and explain the behavior of a failed device using the same kind of knowledge as applied to the device before failure. As long as our knowledge of the domain is sufficiently broad to encompass the failed device, its symptomatic behavior can be explained in terms of its modified physical structure.

Recall that a model in QP theory is composed of two parts: a first-order domain theory, covering the relevant domain(s) for the modeled system; and a propositional scenario description, specifying the physical structure of the system. If we believe that our domain theory is correct, then any flaw in the model must reside completely within the scenario description.

One of the underlying assumptions of this research is that for any given domain, it is possible to enumerate the types of primitive entities and processes needed to completely describe any system within that domain. In the domain of thermodynamics, for example, the relevant processes consist of phase transformations and flows of fluid and heat; while the types of entities include fluid objects, containers, heat sources, paths and pumps. In analog electronics, there is only one type of process (charge flow), and only a few primitive entity types (e.g., resistance, inductance, capacitance, voltage source, etc.) In a digital circuit, there are only 16 possible two-input gates. These examples illustrate that it is generally possible to completely cover a domain at a sufficiently abstract level.

By treating a domain theory as closed (i.e., complete), it is possible to rule out certain behaviors as physically impossible. This is different from closing a set of fault models for a component. A capacitor might conceivably fail in such a way that it behaves as a resistor, or as a capacitor and resistor in series, or as any of an infinite number of possible circuits. Enumerating these possible transformations is generally impossible; however, it is possible to cover the space of the transformations via a closed domain theory. There may be certain constraints which are common to all possible failure modes; for example, no matter how a capacitor fails, it will continue to obey Kirchoff's Laws and the Conservation of Energy Law. These constraints may be used to rule out candidate diagnoses found by the consistency-based approach.

The overall approach taken in this research consists of two steps. In the first step, the assumptions underlying the model are selectively retracted (actually, negated) in search of a symptom-consistent model of the device. This model may not predict or explain the symptomatic behavior, but at least it no longer makes incorrect predictions. This step is called candidate generation, and is the subject of Chapter 4. The second step searches within a particular symptom-consistent model for a more precise model which predicts the observed symptoms. This is the subject of Chapter 5. The details of the approach are presented below.

2.4.1 Qualitative Symptoms

Diagnosis begins with a behavioral description which does not match expectations. These deviations from expected behavior are called *symptoms.* Symptoms are represented qualitatively, and are of one of the following forms:

Inequality Deviation: Some inequality relation was observed to be different than expected; this includes deviations in directions of change.

Magnitude Deviation: Some quantity or rate is observed to be larger or smaller than expected; such deviations may not be detectable from a purely qualitative model, but may still provide useful qualitative information.

Discrete Deviation: Some object or event is unexpectedly absent (or present). In some cases these may be represented as inequality deviations, where the mass of some object or the rate of some process has become zero (or non-zero).

Symptoms may be detected automatically, by comparing the outputs of sensors with the predictions from a numerical or qualitative simulator. Alternatively, symptoms may be provided by a human observer, who is comparing observed behavior with expectations derived from a mental model. This research does not address the issue of how symptoms are obtained; it is assumed that symptoms are provided by some external agent.

2.4.2 Where is the Failure?

A failure is viewed as residing within the model of the device, rather than within the device itself. A broken device continues to obey the natural laws; it is our model of it which has failed to keep pace with the changes undergone by the device. Diagnosis involves a particular kind of failure within the process-centered models of QP theory, as explained below.

2.4.2.1 Failed Theory verses Failed Description

As defined in QP theory, a process-centered model is composed of a general domain theory and a specific scenario description. Failure of a model may be due either to a failure of the domain theory or to a failure of the scenario description. Addressing failures within the domain theory is called *theory revision*, and is also an area of active research in AI [65]. Addressing failures within the scenario description is the chief concern of diagnosis. While many of the techniques discussed in this thesis are applicable to both types of model failures, the emphasis here is on diagnosis.

2.4.2.2 Errors of Omission and Commission

A scenario description may be in error due to either omission or commission. An error of omission occurs when some component or structural relationship present in the physical device is omitted from the scenario description. An error of commission occurs when some entity or relationship stated within the scenario description is in fact not physically present. These two types of errors pose very different problems, in both detection and correction. Errors of commission are generally easier to identify, because the space of possible commissions is finite. On the other hand, the space of possible omissions is infinite, and so can be intractible to search explicitly. One of the contributions of this research is the ability to efficiently search the space of possible omissions from the scenario description.

2.4.3 Repairing the Model

Diagnosis is a search for minimal perturbations to a faulty model of a device, such that the predictions of the model match the observed behavior of the device. This follows the AI paradigm of *generate and test*, whereby candidates are suggested and then checked against a set of requirements. Efficient search requires that the generator make intelligent choices about which paths to consider next. Inconsistent paths should be rejected as early as possible, and the remaining paths should be ordered based on estimates of their probability of success. This has the effect of focusing the search for a post-failure model in the immediate vicinity of the original model. Empirical evidence supports the intuition that for most failures, the device after failure differs only slightly from its original working condition. The minimization of abnormalities to achieve consistency is a recurring theme in default reasoning and nonmonotonic logic, as it is throughout this thesis.

2.4.3.1 Generating Symptom-Consistent Candidates

The simplest approach to generating candidate diagnoses is to randomly apply some perturbation to the model, and then analyze the perturbed model to see if its predictions agree with the observations. However, the presence of a qualitative model allows us to do better than this. Judicious use of the model allows the search to remain focused on reasonable possibilities.

The Role of Processes Processes provide an intermediate representation between structure and behavior. In order for a candidate model to match the observations, its process structure must be consistent with those observations. If some quantity is unexpectedly changing, then there must be some active process influencing it. Similarly, if a quantity is expected to change but does not, then either some known process is unexpectedly inactive, or some other process is interfering with the known process.

The space of possible processes is smaller and more abstract than the space of all possible structures, so it is more efficient to find a valid process structure before speculating on the underlying structural details. In addition, processes can be indexed by their influences, so that only those processes capable of influencing some unexpectedly changing quantity need to be considered.

There are two ways to perturb the process structure of the model: removing an existing process, or adding a new process. These correspond to the failures of commission and omission, respectively, introduced above. Each of these requires a different style of reasoning:

Removing an Existing Process An active process may become inactive as a result of a failure. If removing a process explains an unexpected observation, then that process may no longer be active. If a quantity q is expected to change but is observed to be constant, it is reasonable to suspect that the process responsible for changing q is inactive due to the failure. For example, if a liquid flow was occurring between two containers prior to failure, and after failure both liquid levels are observed to be constant, it is likely that the flow has somehow been halted. The other possibility is that the process is being counteracted by one or more competing processes.

Adding a New Process It is generally more difficult to consider model perturbations involving the addition of previously unknown process instances, since there can be an arbitrary number of them. Still, there are only a fixed number of process *types*, and only those with the appropriate kinds of influences need to be considered. The obvious case for postulating a new process is where a previously-uninfluenced quantity is unexpectedly changing; since all change is caused by processes, there must be some unknown process at work. For example, if the water level in some container is believed to be uninfluenced yet is observed to be dropping, then there must be some process (e.g., a leak) which is negatively influencing the amount of water. In the general case, adding or removing a process could explain why a quantity is changing at a different rate or in the opposite direction than expected.

Modifying the Scenario Description Generating a candidate process structure is an intermediate step toward identifying the underlying structure of the broken device. A new structural description of the scenario must be constructed from the original description, in order to explain the new process structure. The objective is to perturb the original description as little as possible while satisfying the requirements for the new process structure.

Deletions from the process structure generally can be achieved by removing a component from the scenario description, so long as it does not still participate in other ways. For example, a fluid flow process may be eliminated by removing the flow path. This does not imply that the physical pipe has vanished—rather, that the pipe is no longer behaving as a path (perhaps because it is clogged).

Additions to the process structure require identifying components to participate in the new process. These may be new objects which were not previously known to exist; or they may already be part of the scenario description, but filling other roles. For example, in the case of a leaking container, the container itself fills the role of the path for the fluid flow process. We knew about the container—but we did not know that it was also a path. The issue of whether to posit new objects or to reuse existing ones to fill needed roles is addressed in Chapter 5.

2.4.3.2 Relation to Traditional Consistency-Based Diagnosis

The consistency-based approach to diagnosis introduced in Section 2.3 finds candidate diagnoses which are consistent with observed symptoms, by temporarily suspending the constraints associated with suspect components. This research extends this approach in several ways. Traditionally, suspension of a component's constraints only removes inferred behaviors sanctioned by the model; no new behaviors are added. Rather than suspending a component's constraints, in this approach the component itself is "suspended" by temporarily removing it from the model. Removing a component can have nonmonotonic consequences to the model. In particular, processes which depend on the component will no longer exist, and their influenced quantities may be changing in different directions than those predicted by the model. Thus the first extension is a shift from *suspension* of constraints to *negation* of base assumptions underlying the model.

Another significant departure from the traditional consistency-based approach involves the nature of the underlying assumptions themselves. In addition to assumptions about the constituent components and their connectivity, this research considers several other types of assumptions which may require retraction during diagnosis. Most significant among these is the *closed-world assumption*, which is made necessary by the compositional nature of the model. A closed-world assumption is required for the inference that an uninfluenced quantity is constant, for example. Closed-world assumptions and other defeasible assumptions are discussed in detail in Chapter 3.

Candidates generated by the minimal retraction algorithm presented in Chapter 4 may not be acceptable as a structural description of the broken device. Explaining a failure as a closed-world assumption violation does not constitute an acceptable diagnosis, because it does not describe the post-failure structure of the device. Because closed-world assumption violations do not constitute a structural description, candidates containing them must be further refined, as described below.

2.4.3.3 Searching for Explanations: Abduction

Diagnosis involves a search for a model of a broken device, whose predictions agree with the actual observed behavior of the device. Abductive backchaining is one approach to searching for such a model. A domain theory may be represented in terms of rules, whose antecedents are the structural properties of a device, and whose consequents are the resulting behavior predicted for that device.[1]

During model composition, the rules contained in the domain theory are run in a forward chaining manner. Given a set of known facts describing the structure of the scenario, the rules yield predictions about the expected behavior of the device. Facts in the database are matched with rule antecedent triggers, and when successful, rule consequents are instantiated and justified accordingly. Abduction searches backward through these rules, starting with observed behaviors and terminating on a physical structure whose model predicts the observed behavior. Unlike deduction, abduction is not logically sound. It implicitly assumes that all phenomena have a *cause*, and that the model contains the complete set of causes for each phenomenon.

Abduction over a first-order theory can be extremely inefficient, and is NP-hard in the general case [71]. Whereas forward chaining never ventures outside the deductive closure of its knowledge, abductive backchaining typically expends significant resources following dead-ends, trying out explanations which inevitably fail. Thus it is essential that any abductive search be

[1]This is an over-simplification; there are rules whose antecedents and consequents are intermediate between structure and behavior; see Chapter 3.

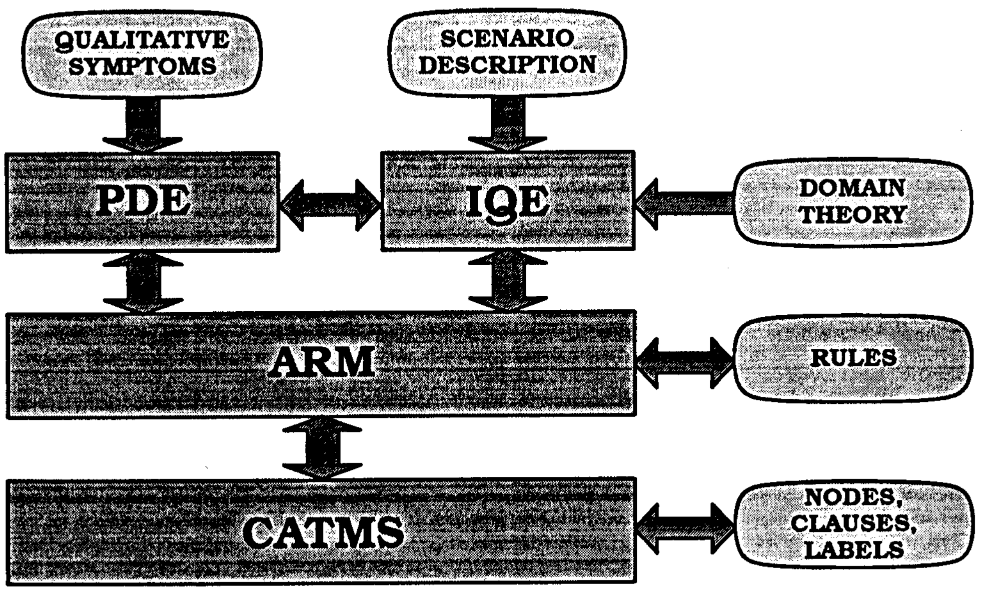


Figure 2.2: Diagnostic System Diagram

narrowly focused on a small portion of the model. In this research, abductive search is limited to finding missing structure, and is guided by a specific symptom-consistent candidate.

The overall approach taken here consists of two steps. The first step utilizes the minimal retraction algorithm presented in Chapter 4 to generate candidate diagnoses, by selectively negating individual assumptions underlying the model until it yields predictions consistent with observed symptoms. The second step is applied to any candidate diagnosis containing a negated closed-world assumption; such candidates are considered to be intermediate results which must be further refined. This refinement is accomplished by performing an abductive backchaining search within the rules defining the domain theory, for an explanation of how each closed-world assumption was violated. This abductive search is the subject of Chapter 5.

2.5 The Process-Based Diagnostic Engine: PDE

The Process-based Diagnostic Engine (PDE) is the implementation of the approach outlined above. PDE is implemented from several components which were developed as part of this research. Figure 2.2 depicts the various modules and their interactions.

CATMS [5, 29] is a variation of deKleer's assumption-based truth maintenance system (ATMS), offering some unique capabilities; it is described briefly in Chapter 4. ARM is the rule engine providing the interface to CATMS, and is discussed in Chapter 3. IQE [30] is the incremental qualitative envisioner upon which qualitative models are constructed; its modeling language is presented in Chapter 3. These components have general utility in other research domains, and are the result of a joint effort with Dennis DeCoste.

The diagnostic engine is the top-level module in the overall system. It controls the other modules and provides all input and output functions. Inputs to the system consist of a structural description of a device in its pre-failure condition, a set of qualitative symptoms (as described in Section 2.4.1), and a qualitative domain theory covering the given scenario.

The domain theory and scenario description are given to IQE, which translates the domain theory into a set of rules in ARM, and then feeds the scenario description to these rules. The result is a set of nodes and justifications in CATMS representing the qualitative constraints of the domain theory as they apply to the given scenario. This justification structure and the resulting node labels provide the foundation for subsequent reasoning.

The basic algorithm implemented by PDE is presented below. It starts by instantiating the model and then checking the consistency of the set of base beliefs (*Base_Env*), consisting of the scenario description ($\mathcal{SD}$), the modeling assumptions ($\mathcal{A_M}$), the closed-world assumptions ($\mathcal{CWA}$s), and the observations ($\mathcal{OBS}$). If an inconsistency is detected, PDE initializes the minimal retraction algorithm and then calls it repeatedly looking for candidate explanations. If a candidate containing a closed-world assumption is found, it is passed to the abduction algorithm for further refinement. The algorithms for *Min_Retract*() and *Abduct*() are the subjects of Chapters 4 and 5, respectively.

```
Program PDE($\mathcal{DT}$, $\mathcal{SD}$, $\mathcal{A_M}$, $\mathcal{OBS}$)
    call Instantiate_Model($\mathcal{DT}$, $\mathcal{SD}$);
    let Base_Env = $\mathcal{SD} \cup \mathcal{A_M} \cup \mathcal{CWA}s \cup \mathcal{OBS}$
    if Consistent(Base_Env, $\mathcal{DT}$) then
        return "Observations Consistent";
    call Init_Min_Retract();
    repeat
        let $E_i$ = Min_Retract(Base_Env, $\mathcal{DT}$);
        let CWAs = $Neg(E_i) \cap \mathcal{CWA}s$;
        if CWAs = $\emptyset$ then
            add $E_i$ to CANDIDATES;
        else call Abduct($E_i$, CWAs);
    until acceptable candidate is found.
```

2.5.1 Generating Candidate Diagnoses

The first step in diagnosing a failure is to understand the nature of the conflict between the predictions of the original model and the observed symptoms. Specifically, those aspects of the model leading to the failed expectations must be identified. By focusing on the conflicts between prediction and observation, we remove from suspicion those portions of the model which continue to yield valid predictions of behavior.

2.5.1.1 Finding Consistent Models

The consistency-based approach to diagnosis identifies subsets of the model which are consistent with observed behavior. Of course, the "null model", consisting of no component constraints at all, will always be consistent with any behavior. Because PDE negates assumptions rather than merely retracting them, it will never consider the null model or in fact any model smaller than the original one.[2] In diagnosis we are interested in *minimal* deviations from the original model which are consistent with observations.

Chapter 4 presents an algorithm for finding minimal deviations from a set of base assumptions which remove an inconsistency. The algorithm is an extension of the *hitting set* algorithm proposed by Reiter [68] and the consistency-based approach of GDE [25, 26] and *Sherlock* [27]. In addition to improving the efficiency of Reiter's algorithm, the extension involves a change from retracting an assumption to negating it, replacing the removed assumption with its boolean negation. This distinction is significant because in a process-centered model, a negated belief about the presence of a component often results in nonmonotonic additions to the predictions of the model. A component is either present or not; negating the assumption for a component corresponds to physically removing the component from the system. The model produced by the traditional consistency-based approach is much weaker, as it makes no claim as to whether the component is present or not, or what might be in its place.

In GDE, assumptions always represent the presence of a functioning component. In process-based diagnosis, there are several other types of assumptions, including modeling assumptions and closed-world assumptions, which can show up as culprits. Violation of a modeling assumption suggests that a change in perspective might yield consistent predictions. By allowing a closed-world assumption to be violated, the system can hypothesize missing structure and resulting active processes which were previously unknown. The full set of defeasible assumptions is described in Section 3.6.

2.5.2 Candidate Completion

Candidate diagnoses generated by the above approach are not guaranteed to be physically realizable—especially when they involve a closed-world assumption violation. In order to validate such candidates, it is necessary to search for some physical manifestation explaining the violated closed-world assumption. This is done using the abduction algorithm presented in Chapter 5. Given a violated closed-world assumption, PDE performs an abductive search over the rules defining the domain theory, looking for a physical structure which explains the violation. This is how the gaps in the scenario description are filled in, thereby completing the model.

2.6 Discussion

This chapter has provided an overview of process-based diagnosis, including the two elements from which it was conceived: model-based diagnosis and Qualitative Process theory. The ideas and claims introduced here are reinforced and clarified throughout the rest of the thesis.

When observed behavior conflicts with the predictions of the model for a device, the failure is viewed in this research as residing in the model, rather than the device itself. The model has failed to keep pace with the structural changes undergone by the device. Because the device continues to obey the laws of nature, we can understand its symptomatic behavior using the same knowledge which applied to the device before failure.

This research is about diagnosing and repairing a faulty model. A process-centered model is composed from a domain theory and a scenario description; thus any model failure must reside

[2]Measured as the cardinality of the set of base assumptions.

in one or the other (or both). This research focuses on diagnosis; therefore it is assumed that the problem lies in the scenario description. Thus repairing the model involves a search for a minimal perturbation to the scenario description which results in eliminating all symptomatic predictions. The search is guided by the specific nature of the conflicts between prediction and observation, which suggest promising perturbations to the model. Because the domain theory is not brought into question, it is available to focus the search for the underlying physical structure of the failed device.

The compositional nature of process-centered models requires explicit closed-world assumptions to be included in the set of base beliefs which may need to be retracted to repair the model. Closed-world assumptions and their violations are discussed at length in Section 3.5. A violated closed-world assumption indicates that the scenario description is incomplete, because the failure has created new structure. Closed-world assumption violations are transformed into complete scenario descriptions using abductive search through the rules of the model, as described in Chapter 5. This unique combination of consistency-based and abductive approaches to model-based diagnosis combines the efficiency of the former with the inferential power of the latter; this is one of the identifying features of process-based diagnosis.

Process-based diagnosis places a great burden on the domain theory, which must be sufficiently general to cover a device after a failure occurs. In effect, the breadth of the domain theory replaces the explicit fault models of earlier approaches. As the diversity and accuracy of domain theories continue to improve, process-based diagnosis will come to realize its true potential.

Chapter 3

Modeling for Diagnosis

3.1 Introduction

Model-based diagnosis relies on a model of a device to explain its failures. A model serves two primary roles in diagnosis. First, it yields predictions about the expected behavior of the device. As long as the predicted and actual behaviors match, there is no reason to suspect that a failure has occurred. When the predicted behavior fails to match the actual observed behavior, then a failure has occurred and the diagnostic process begins. The more significant role of the model is to explain why a failed prediction was believed. By identifying the components contributing to a failed prediction, the diagnostician can focus on the most likely region of failure.

As discussed in the previous chapter, a failure is viewed as residing within the model of a device rather than within the device itself. Explaining a failure thus amounts to identifying and repairing the portions of the model whose predictions no longer match the observed behavior of the device. The methods used to accomplish this are the subject of Chapters 4 and 5.

This chapter discusses the modeling issues which have been addressed by this research. Section 3.2 highlights the infrastructure upon which models are constructed. Section 3.3 describes the modeling language used for defining domain theories and domain-independent constraints. Section 3.4 defines the domain-independent constraints sanctioned by qualitative reasoning and QP theory. Section 3.5 discusses default reasoning and the explicit representation of closed-world assumptions, which are violated when a failure introduces new structure. Section 3.6 describes the other classes of defeasible assumptions considered in this research. Finally, Section 3.7 summarizes the contributions of this research in the area of modeling for diagnosis.

3.2 Modeling Infrastructure

Searching for diagnostic explanations involves both forward and backward reasoning on the constraints defining the model of the failed device. This requires a uniform representation for both the domain theory and the domain-independent qualitative constraints which underlie qualitative reasoning. Whereas qualitative simulators often take a procedural approach to implementing these constraints, here it is essential that all constraints be expressed uniformly as rules.

The term *model* as used in the literature has a wide variety of meanings. In this research, model refers to the combination of a general first-order domain theory and a specific propositional scenario description, as defined in QP theory [37]. A domain theory may be decomposed into individual *model fragments*, each of which is a partial description of some entity, relationship, physical law, or phenomena. The model fragments of a domain theory may be represented as rules defining the behavior of minimal assemblies of components required to instantiate active processes. A scenario description consists of a set of propositions specifying the structural details of a single device, including its component parts and the connectivity between them. Because the elements of a domain theory are *compositional*, components may be assembled into arbitrarily large systems, allowing a single domain theory to cover an unbounded number of distinct scenarios.

Certain constraints are common across all domains, and may be represented as a set of domain-independent rules. The domain-independent rules together with a particular domain theory form the *rule base* for the model. The propositions contained in a scenario description constitute a portion of the *database* of the model. The database does not support patterns containing variables; it is purely propositional. As rules in the rule base fire on these propositions, additional derived propositions are created and added to the database.

3.2.1 Conventions

The following conventions are used throughout the thesis. Variable names consist of a "?" followed by one or more alphanumeric characters—for example: $?Q$, or `?CAN3`. `Typewriter font` is used for actual code or code fragments, and *italics* is used for purely conceptual constructs. Constants are shown in UPPER-CASE, predicate names are Capitalized, and function names and logical connectives are set in lower-case. Positive literals may be represented by a single capital letter (possible subscripted) (e.g., P_i), or written out as a predicate followed by one or more arguments, as in $Mammal(FRED)$ (conceptual) or `(Container ?Can3)` (code fragment).

3.2.2 Assumption-Based Truth Maintenance

Given the view of diagnosis as a search through the space of models, it is essential that we have some way of representing and manipulating multiple models efficiently. It is natural to view a candidate model as a set of assumptions, where each assumption represents a particular default belief (such as the proper functioning of some component). An assumption-based truth maintenance system (ATMS) provides an efficient mechanism for reasoning with multiple situations simultaneously, thereby avoiding duplication of effort inherent in most backtracking schemes.

This research has evolved around the use of a hybrid ATMS which is described in Section 4.4. The ability of the ATMS to move freely among mutually inconsistent contexts is central to the process-based approach to diagnosis. This section highlights the key concepts of the ATMS which are essential to understanding the rest of this thesis.

3.2.2.1 Nodes, Assumptions and Environments

Briefly, an `ATMS` can be characterized as follows. Propositions are represented as ATMS *nodes.* Logical constraints among nodes are represented as *justifications* or disjunctive *clauses.* Certain nodes are designated as *assumptions*, indicating that we wish to entertain their truth as a possibility. Each node has an associated *label*, indicating the minimal sets of assumptions, or *environments*, in which it is true.

Environments may be interpreted as situations in time. Nodes in an ATMS typically do not refer to a particular situation, but to all situations. The truth of a proposition may only be ascertained given the context of a particular situation, represented as an environment. An alternative interpretation of an ATMS environment is that of a possible world. Many possible worlds may be entertained simultaneously. A node may be true in one possible world, false in another, and unknown in a third. This implicit representation of time is quite different from more traditional representations where time is represented explicitly (such as the situation calculus), and takes some getting used to. Because of the way an ATMS caches the results of its reasoning, the overhead of context switching is minimal. This is the most significant advantage of using an ATMS.

Because any nontrivial model describes a dynamic system, the set of true propositions will vary from one situation to the next. The database contains all propositions relevant to the model, not just those which are true in any given situation. An ATMS generally does not support arbitrary retraction of propositions; only assumptions may be retracted. Thus assertions of invariant truth must be made with caution. Propositions may be partitioned into three types:

Invariants Propositions which are asserted to be true in every situation.

Assumptions Propositions which are assumed to be true in some situations or possible worlds.

Deductions Propositions whose truth depends logically on one or more other propositions.

The nature of diagnosis is to question the structural description defining the model; propositions from the scenario description which would otherwise be asserted as invariants in a qualitative simulation must be assumed to allow for retraction during the diagnostic process. The only propositions which remain unchallenged during diagnosis are invariants from within the domain theory itself. Were we to expand our scope to include theory revision, then even these would be subject to retraction and would have to be assumed.

3.2.2.2 Clauses, Justifications and Incompleteness

Logical constraints among propositions are naturally expressed as implications, such as:

$$Man(SOCRATES) \Longrightarrow Mortal(SOCRATES)$$

Such constraints among propositions are typically called *justifications* in the TMS literature [16, 17, 18]. In general, a justification for a proposition P is a logical relation of the form: $\bigwedge_{i=1}^{n} A_i \Longrightarrow P$; each proposition A_i is called an *antecedent* of the justification, and P is called the *consequent.*

Alternatively, constraints may be expressed as disjunctive clauses, such as:

$$Animal(ITEM) \vee Vegetable(ITEM) \vee Mineral(ITEM)$$

Sets of such clauses are said to be in conjunctive normal form (CNF). Logically clauses and justifications are equally expressive; one is easily transformed into the other. A sound and complete theorem prover would treat both forms identically. As a result the terms are often used interchangeably.

The distinction becomes relevant for incomplete reasoning techniques, such as forward propagation through justifications, or boolean constraint propagation (BCP) through clauses. BCP is known to be complete for Horn theories, but not in the general case. These techniques are at the heart of label propagation in most ATMS implementations, including the one used in this research. A typical ATMS might only reason forward through a justification, thus making it less complete than the corresponding clausal representation. In this research, clauses are used exclusively within the ATMS in lieu of justifications. This is more complete than forward reasoning through justifications, but is still potentially incomplete. Full resolution of clauses is required to achieve completeness in an ATMS, and this is prohibitively expensive. This research strives for efficient reasoning by working with sound but incomplete reasoning techniques like BCP.

For simplicity of discussion, the term justification is used in this thesis to refer to what is internally represented as a CNF clause. A proposition is said to have *incoming* and *outgoing* justifications; incoming justifications correspond to clauses containing the proposition, and outgoing justifications correspond to clauses containing its negation. By definition, a node is the consequent of its incoming justifications, and is an antecedent of its outgoing justifications. Justifications typically arise through the instantiation of first-order rules, as discussed below.

3.2.3 Modeling with Rules

Domain theories and domain-independent constraints may be defined in terms of rules. In its most general form, a rule contains a list of patterns, called *triggers*, and a *body* containing one or more actions to be performed. When the triggers match propositions in the database, the rule *fires*, causing the actions in the body to be carried out.

Triggers typically contain variables, which become bound when the triggers unify with propositions in the database. These variable bindings are applied to the body of the rule upon firing. Each combination of propositions matching the triggers results in a separate firing of the rule and a separate execution of the rule body.

3.2.3.1 Implication Rules

Most rules express a logical relation between a set of propositions. Typically the presence of a certain pattern of antecedent propositions sanctions the inference of one or more consequent propositions. Rules of this form are called *implication rules*, and are represented using the following syntax:
`(==>` *<antecedent> <consequent>*`).`
The antecedent defines the trigger for a forward-chaining rule. Variables in the antecedent are bound when the trigger unifies with propositions in the database. Bindings from the antecedent are applied to the consequent, and the result is instantiated and justified by the antecedent. Figure 3.1 illustrates the instantiation of an implication rule, and the resulting justification. Because the database is purely propositional, the consequent of a forward-chaining rule must not introduce any variables not bound in the antecedent.

The logical connectives `and`, `or` and `not` are fully supported. A conjunctive antecedent introduces multiple triggers which must all match before a rule fires. A disjunctive antecedent is decomposed into separate rules, one for each disjunct. Likewise, a conjunctive consequent is decomposed into separate justifications within a rule. Disjunctive consequents are implemented

Rule Base:

```
(==> (Container ?can) (Positive (volume ?can)))
```

Knowledge Base (before):

```
(Container CAN34)
```

Justification:

```
(justify (Positive (volume CAN34))
         ((Container CAN34)))
```

Knowledge Base (after):

```
(Container CAN34)
(Positive (volume CAN34))
```

Figure 3.1: Instantiation of an Implication Rule into a Justification

as clauses containing the additional disjunct of the negated antecedent. Negation is automatically propagated down to the literals. For example, the rule:
`(==> (A ?x) (and (B ?x) (not (and (C ?x) (D ?x)))))` and the fact: (A 1) would result in the following clauses:

```
(or (not (A 1)) (B 1))
(or (not (A 1)) (not (C 1)) (not (D 1)))
```

Rules may also contain the connectives `iff` and `==>`, which are implemented by the appropriate disjunctive clauses. The actual implementation of logical connectives is handled by a model translator, as described in Section 3.2.3.4.

3.2.3.2 Rule Keywords

Three keywords are interpreted specially within rules. These keywords are represented as functions which modify the interpretation of their arguments. The first keyword, `:TEST`, may appear only within a conjunctive antecedent, introducing the second or later conjunct. The single expression following the `:TEST` keyword is interpreted as LISP source code, after performing variable substitution; the result is used as a Boolean condition on the rule's firing. If the expression returns non-nil, rule execution proceeds normally; otherwise the rule is blocked from firing on the given proposition(s).

The remaining two keywords: `:ASSERT` and `:EVAL`, may occur only within the consequent. The `:ASSERT` keyword signals that the consequent is to be made true regardless of the truth of the antecedent; the justification created by the rule has an empty antecedent. The `:EVAL` keyword signals that the expressions following it are to be evaluated as LISP code (after substituting for variables) upon firing of the rule. While the majority of rules do not require any of these keywords, their inclusion greatly enhances the flexibility and representational power of the rule system.

3.2.3.3 A Note on Rule System Implementation

The rule system implemented in ARM is a derivative of the ATMoSphere rule system developed for use with the ATMS [43]. As in ATMoSphere, conjunctive antecedents in ARM result in a chain of rule triggers, one for each conjunct. Each firing of the first rule trigger results in a separate binding for the second trigger, and so on. Rule trigger–binding pairs hand off control from one to the next, until the last trigger hands off control to the rule body.

Rule triggers containing unbound variables are implemented using a simple indexing scheme, whereby rule triggers and propositions are partitioned into classes by predicate. New rules are checked against existing propositions, and new propositions are checked against existing rules. This scheme works well when predicate classes contain few rules and propositions, but can perform poorly on large classes associated with common predicates such as >=. In practice, most rules on these classes are intended to fire on every proposition in the class, so no efficiency is lost.

Fully instantiated rule triggers (i.e., those containing no unbound variables) are handled specially. If the corresponding proposition already exists, then the rule is fired immediately. Otherwise a *phantom proposition* is built as an index to the rule, and marked to distinguish it from ordinary propositions. The phantom proposition is installed in the proposition hash table. If the (real) proposition is eventually instantiated, the phantom is accessed and the rules indexed there are run. This indexing scheme significantly reduces the overhead of the rule system as compared to the predicate class indexing approach. This is especially true because most of the rule-binding pairs involve the later triggers of multiple-trigger rules, and these are exactly the triggers which are most likely to have all of their variables bound by previous triggers. Based on empirical results, better than two thirds of the rule-binding pairs are fully instantiated and so are either applied directly or cached on phantoms; run-times for the rule system alone show a twenty percent improvement under this scheme.

3.2.3.4 Translation Rules

For efficiency reasons, some portions of the available modeling language are recognized as superfluous, in that they do not add to the expressiveness of the language. For example, all of the following are legal expressions within the modeling language, indicating that a quantity Q is positive:

```
(Positive Q)          (Pos Q)
(Greater-Than Q 0)    (Less-Than 0 Q)
(> Q 0)               (< 0 Q)
(not (<= Q 0))        (not (>= 0 Q))
```

Such constructs are included in the language for convenience and for compatibility with older models. However, including them in the internal representation and relating them to their equivalent forms would incur a stiff penalty in run-time performance. For this reason, these redundant modeling constructs are converted at compile-time to their canonical internal forms (the last form in the example above).

A *translator* is applied recursively to the antecedent and consequent forms of rules during compilation. The translator is an integral part of the rule system which checks each encountered predicate for an associated *translate-function*. If one is found, it is applied to the containing proposition (i.e., the predicate and its arguments), which is replaced by the result of the translation. If this result differs from the original form, then the translator is applied again. The recursion stops when no translate-function is found, or when application of the translate-function returns its input form unmodified. A translate-function is called for its result only; it can have no side-effects.[1] The translator allows domain theories, scenario descriptions and domain-independent constraints to be specified more naturally, without affecting the underlying internal representation.

Translate-functions are defined using a syntactic variation of the implication rule, called a translation rule. A translation rule has the following syntax:
`(=TR=>` *<pattern>* *<translation>*`)`.
Conceptually, translation rules may be interpreted as ordinary implication rules, despite their very different implementations. The distinction is important for efficiency reasons only. Translations are applied during compilation of ordinary rules, replacing intermediate patterns with their equivalent translated forms. The consequent of a translation rule may itself be an intermediate form requiring further translation and thus never appear in the database.

3.2.3.5 Backward Chaining

Every implication rule is indexed both for forward and backward chaining. The forward chaining rule system was described in Section 3.2.3.3. Rules are indexed for backward chaining based on the consequent of the rule, using the same predicate indexing scheme used for forward chaining. Rules having disjunctive consequents are indexed once for each disjunct. However, rules are not indexed for backchaining on the negated antecedents of the rule, even though logically a negated antecedent is indistinguishable from the consequent in the resulting CNF clause. The rules are directional, even though the resulting clauses are not; this is true for both forward and backward reasoning.

Backward chaining is used in this research to identify missing structure, which is detected as a closed-world assumption violation (described in Section 3.5.1). A candidate diagnosis containing a negated closed-world assumption is further refined by searching backward through the rules for an explanation for the violation. Backward chaining is applied recursively until the search grounds out in a set of structural constraints suitable for inclusion in a scenario description. Chapter 5 explains the abduction process in detail.

3.3 The Modeling Language

The modeling language described here is based heavily on that of the Qualitative Process Engine `QPE` [40]. The modeling language of `QPE` was developed with the primary goal of capturing the commonsense intuitions of the person on the street. The emphasis in diagnosis is of a more technical nature, as its models must capture the reasoning of an experienced engineer or diagnostician. This shift in emphasis together with lessons learned from years of modeling suggest some enhancements to the modeling language. Many of the changes amount to syntactic sugaring for improved readability and modeling convenience. Other changes evolved from the need for an explicit, inspectable representation of internal operations which were previously implemented procedurally. Still other changes—such as the move to soft inequalities—were

[1]This restriction has not presented a problem in the models developed thus far.

```
;;; <n> is a number; <q> is a quantity; <op> is one of {+,-,*,/}:

(Quantity <q>)

(>  <n> <n>)    (<  <n> <n>)    (=  <n> <n>)
(>= <n> <n>)    (<= <n> <n>)    (!= <n> <n>)

(Pos <q>)   (~Pos <q>)   (Inc <q>)   (~Inc <q>)
(Neg <q>)   (~Neg <q>)   (Dec <q>)   (~Dec <q>)
(Zip <q>)   (~Zip <q>)   (Con <q>)   (~Con <q>)

(:= <q> <q>)    (:= <q> (<op> <q> <q>))

(Same_Sign <n> <n>)    (Opp_Sign <n> <n>)    (Same_Rel (<n> <n>) (<n> <n>))

(Sum_Member <q> <q> <s>)    (Deriv <q> <q>)

(I+ <q> <q>)    (Qprop+ <q> <q>)    (M+ <q> <q>)
(I- <q> <q>)    (Qprop- <q> <q>)    (M- <q> <q>)
```

Figure 3.2: Summary of Modeling Language Constructs

made for efficiency of internal representation. The result is a compact and expressive language for expressing qualitative modeling constraints.

3.3.1 Modeling Constructs

Figure 3.2 summarizes the fundamental constructs of the modeling language. These provide the building blocks from which domain theories, domain-independent constraints, and scenario descriptions are built. Higher-level constructs used for defining domain theories are described in Section 3.3.2.

3.3.1.1 Quantities

The fundamental unit of modeling is the quantity. Quantities represent real-valued parameters which vary smoothly and continuously over time. Quantities may be related in various ways; for example, one quantity may be defined as the difference between two other quantities. At any given instant in time, a quantity has a value. Throughout this discussion, time is left implicit, and a quantity is interpreted to refer to its value in the current instant.

Quantities are modeled as functions of one or more arguments representing the object(s) to which a quantity belongs. The name of the function indicates the aspect of the object(s) represented by the quantity. For example, `(volume CAN1)` represents the the volume of the container `CAN1`. A quantity may be declared with the predicate `Quantity`, as in `(Quantity (volume CAN1))`.

Time-varying quantities have an associated temporal derivative, representing the instantaneous rate of change of the quantity with respect to time. The derivative of a quantity `Q` is represented as (D Q). Conceptually the derivative of a quantity is another quantity; however in practice derivatives are distinguished from ordinary quantities to avoid infinite introduction of higher-order derivatives.

3.3.1.2 Inequalities

Quantities and their derivatives are called *numbers*. There are a few special numbers—such as the integers 0 and 1—whose values are constant and known with precision. Numbers represent real values, and so define a natural ordering. The actual value of a quantity or its derivative may not be known precisely, but may be constrained by a partial ordering over the set of numbers. Ordinal relations between pairs of numbers specify known relations which define this partial ordering, and are referred to as *inequalities* or *inequality relations.*

Inequalities are the principal constraint available in qualitative reasoning. Certain inequalities, such as the *quantity conditions* of views and processes, define the qualitative state of the modeled device. Determining how these inequalities change over time is the principal goal of qualitative reasoning.

The modeling language supports both hard ($>$) and soft ($\geq$) inequality relations. The following inequality predicates as well as the longhand versions in parentheses are supported by the language, taking two numbers as arguments:

```
>  (Greater-Than)      >=  (Greater-Equal)
<  (Less-Than)         <=  (Less-Equal)
=  (Equal-To)          !=  (Non-Equal)
```

Inequalities involving the special number: 0 (zero) are of particular interest in qualitative reasoning, as they define the signs of quantities and their derivatives. The sign of a derivative indicates the direction of change for the corresponding quantity, which determines whether the quantity is moving toward or away from some fixed landmark. In fact much research in qualitative reasoning [54, 79] has been restricted to reasoning about signs.

The signs of quantities and their derivatives may be expressed in a variety of ways. In addition to the inequality predicates above, the following sign predicates and their corresponding longhand versions in parenthesis are defined for quantities:

```
Pos (Positive)   ~Pos (Non-Positive)   Inc (Increasing)   ~Inc (Non-Increasing)
Neg (Negative)   ~Neg (Non-Negative)   Dec (Decreasing)   ~Dec (Non-Decreasing)
Zip (Zero)       ~Zip (Non-Zero)       Con (Constant)     ~Con (Non-Constant)
```

3.3.1.3 Algebraic Relations

New quantities may be defined in terms of existing quantities, using the equivalence operator ":=". In its simplest form, one quantity may be defined to be equivalent to another, as in (:= Q1 Q2). This statement is stronger than merely stating that two quantities are equal, as it also equates all corresponding higher-order derivatives as well. Whereas equality typically holds for only an instant, a defined equivalence generally holds across all situations.

Quantity definitions may include simple algebraic operations. The syntax for an algebraic relation is: (:= Q1 (<*op*> Q2 Q3)), where <*op*> is one of the four binary operators {+,-,*,/}. Arbitrary combination of operations is not supported.

3.3.1.4 Constraints Between Ordinal Relations

An important class of constraints involves relationships among pairs of ordinal relationships. For example, one might know that two numbers always have the same sign, or that one inequality relation is causally tied to another. Constraints among inequality relations are represented using the `Same_Rel` predicate, which takes as arguments two pairs of related numbers, as follows: `(Same_Rel (N1 N2) (N3 N4))`. This states that the relation between `N1` and `N2` must be the same as the relation between `N3` and `N4`.

Given the prevalence of inequalities involving `0`, a special syntax is provided for use when both pairs of numbers contain `0`. The expression `(Same_Sign N1 N2)` enforces the obvious constraint that `N1` and `N2` have the same relation with `0`. Similarly, `(Opp_Sign N1 N2)` constrains the relation between `N1` and `0` to be the same as the relation between `0` and `N2`.

3.3.1.5 Sums

The compositional nature of a general domain theory comes from the ability to dynamically combine the effects of multiple cooperating and competing processes in order to resolve their net effect. Multiple influences on a quantity have a cumulative effect, and so must be added together. Because sums must be supported in order to handle influences, it is a simple matter to generalize the mechanism to handle arbitrary sums.

Sums are represented as sets of constituent quantities partitioned into two groups: those contributing positively to the sum, and those contributing negatively. This distinction is based not on the actual sign of the constituent quantity, but on the direction of contribution on the sum assuming the constituent quantity is positive. In other words, a general sum represents the algebraic difference between the respective sums of its positively and negatively contributing constituents. The purpose for including negative constituents in sums is to support negative influences on quantities, as the next section shows.

Sums are defined in a piece meal fashion—that is, one constituent at a time—using the three-place predicate `Sum_Member`. For example:

```
(Sum_Member (net_force BLOCK27) (force SPRING34) +)
```

indicates that the spring contributes a positive force on the block (whenever the force in the spring is positive). In effect, this statement relates the sign conventions for what constitutes a positive force in the spring and on the block. The following section shows how sums are used for handling influences.

3.3.1.6 Direct Influences

According to a fundamental tenet of Qualitative Process theory, all change may be traced directly or indirectly to one or more active processes. Processes introduce direct influences on quantities, causing them to rise or fall. For example, in the bath tub shown in Figure 3.3, the flow process resulting from turning on the faucet directly influences the amount of water in the tub, causing it to rise. A second flow into the tub would add its effect (and its flow rate); a flow out would subtract from the overall net effect.

Direct influences, which may only be introduced by processes, are represented using the predicates `I+` and `I-`. The bath tub example above could be represented as:

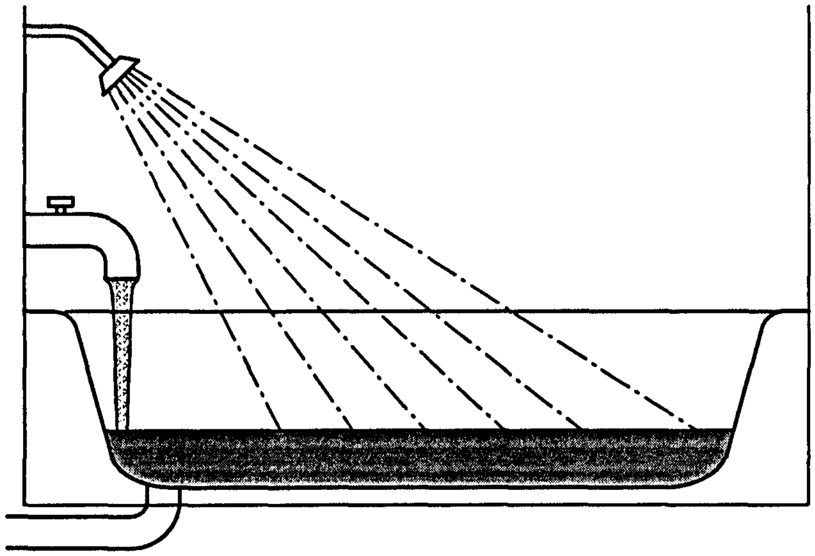


Figure 3.3: Incoming and Outgoing Flows for a Bath Tub

`(I+ (mass (liquid_in TUB5)) (flow_rate FLOW23))`.[2]
Flows occur between two reservoirs, so influences from flow processes generally come in pairs—a positive influence on the destination and a negative influence on the source of the flow.

Influences are represented using the sum mechanism introduced above. The rates of all processes directly influencing a given quantity `Q` are declared members of a common sum quantity: `(net-influence Q)`. Positive influences (`I+s`) introduce positive members of the sum, and negative influences (`I-s`) introduce negative members. The net influence quantity represents the derivative of `Q`—a fact explicitly represented as: `(Deriv Q (net_influence Q))`.

In certain cases the derivative of a directly-influenced quantity has already been represented in the model by an explicit quantity. For example, a domain theory for free body dynamics might refer to the velocity of a moving object. In a process model, velocity defines the rate of a motion process. This process differs from the flow example above in that there can be only one instance of it acting on a given object; there is no possibility of competing or cooperating motion processes. In cases where the derivative of a directly-influenced quantity is explicitly represented, the derivative relationship may be expressed using the `Deriv` predicate, as in `(Deriv (position BLOCK1) (velocity BLOCK1))`. This provides a convenient shorthand for the alternative process definition for the motion process.

3.3.1.7 Indirect Influences

A process can influence a quantity indirectly, through chains of *qualitative proportionalities*, or qprops. A change in one quantity may contribute to a change in another. In some cases we may not know the exact functional relationship between two quantities; still, we may know

[2] Actual domain theories for fluids typically use a more complicated notation for contained fluids; see Chapter 6 and Appendix C.

the direction of influence one quantity exerts on another. Mathematically speaking, we may know the sign of the partial derivative of one quantity with respect to the other. Qualitative proportionalities allow us to express this information incrementally, distributed across various portions of a domain theory specification. This is especially useful during theory formation, when the exact functional relations and even the number of contributing quantities involved are unknown.

When the function relating two or more quantities is well-understood, the ability to compose qualitative proportionalities is not required, and in fact is blocked. For example, a quantity defined as the difference between two other quantities cannot also be influenced by a fourth quantity. The two predicates `M+` and `M-` are borrowed from `QSIM` [54]; they behave like a single qprop, but also block out any other influences on the quantity. Assignments using `:=` and the `Deriv` predicate also bypass the usual mechanism for resolving indirect influences; this is discussed further in Section 3.4.

3.3.2 Domain Theory Constructs

The high-level constructs used for defining domain theories are basically those of `QPE`, with a few minor enhancements. The implication rules described above replace the `defPredicate` syntax of previous models. Also, views and processes support a few additional keywords, as described below.

Views and processes in QP theory are collectively known as conditionalized descriptions; they differ only in that processes contain direct influences, and views do not. For conciseness, the following discussion refers to views, but applies equally to processes.

A view definition has the following general form:

```
(defView  (<name> . <arg_list>)
  Instance_of <CD_list>
  Individuals <Ind_list>
  Model-Asns  <Asn_list>
  Conditions  <Cnd_list>
  Relations   <Rel_list>)
```

The pattern: `(<name> . <arg_list>)` is called the view *signature*; `<arg_list>` includes variables whose bindings for each instance must be unique. A view definition is converted at compile time into a single implication rule; its antecedents are defined by the `Instance_of`, `Individuals`, `Model_Asns` and `Conditions` fields; its consequent is the conjunction of the `Relations` plus the view signature. A simplified depiction of the resulting rule is as follows:

```
(==> (and . <CD_list>  <Ind_list>)
     (and (==> (and . <Asn_list>  <Cnd_list>)
               (<name> . <arg_list>))
          (==> (<name> . <arg_list>))
               (and . <Rel_list>))
```

Each field is optional; if present, a field contains a (possibly empty) list of patterns. The first two fields: `Instance_of` and `Individuals`, define the participating objects and conditions for instantiation. The `Instance_of` field contains the signatures of other views. The individuals field defines the type and conditions of each object. Each individual entry has the general form: `(<var> :TYPE <pred> :TEST <lisp> :FORM <pattern> :CONDITIONS . <conds>)`; again, each keyword is optional. Conditions and the type predicate translate into ordinary triggers for the resulting rule; the :FORM entry allows substitution of a variable for a complex pattern; the variable may be used anywhere in the definition of the view. The Model_Asns and Conditions are handled similarly within the rule; Model_Asns define which segments of a domain theory are to be used, while Conditions contain preconditions and quantity conditions of the view, as defined in QP theory.

Chapter 6 provides examples of view definitions using this syntax, and Appendix A provides a complete listing for the thermodynamics domain theory.

3.4 Modeling Domain-Independent Constraints

This section states in simple rules what is often buried in opaque LISP code in other qualitative reasoning programs. Having a uniform representation for all aspects of the model is essential for the abductive account given in Chapter 5. The set of rules implementing the domain-independent constraints of qualitative reasoning and QP theory are relatively few in number, and the entire set is included here.

Figure 3.4 contains rules for implementing inequality constraints. The first set of translations provide alternate representations for the signs of quantities and their derivatives. The predicates Pos, Neg, Zip, Inc, Dec, and Con along with their negative counterparts are translated into inequalities involving 0. Because of the pervasiveness of signs in qualitative reasoning, these sign predicates are used quite extensively. Inequalities are represented internally using the single predicate >=, thereby greatly simplifying the machinery necessary for handling inequalities. The next rule enforces the anti-symmetric nature of strict inequalities—one number cannot be both greater and less than another number. Internally this constraint translates into a single disjunctive clause for each inequality relation (i.e., pair of related numbers).

The Same_Rel predicate in Figure 3.5 results in two iff clauses enforcing the equivalence of the two inequality relations. Two numbers may be constrained to have the same signs or opposite signs; these are translated into special cases of the more general constraint relating two pairs of related numbers, using the Same_Rel predicate.

Figure 3.6 lists the rules for implementing open-ended sums. The predicate Affected is used internally to represent the presence of a positively or negatively contributing member of the sum. For example, if a sum is affected positively but not negatively, then the sum is positive; if the sum is not affected positively, then it cannot be positive. Sums utilize qualitative proportionalities to relate the derivative of the sum to that of its contributing members.

Figure 3.7 lists the rules for handling direct and indirect influences. The Influenced predicate is analogous to the Affected predicate above, except that it relates to the derivative of the influenced quantity. For example, if a quantity is influenced positively but not negatively, it must be increasing; if it is not influenced positively, then it cannot be increasing. The Influenced predicate is inferred for a quantity having a qualitative proportionality with a non-constant influencer; the sign is determined from the sign of the qprop and the direction of change of the influencer. Direct influences are handled using the sum mechanism discussed above, defining the net influence on the quantity as the sum of the direct influences. The Deriv predicate completes the connection.

Figure 3.8 contains the rules implementing the four algebraic operators: {+,-,*,/}. The rules take full advantage of the symmetry of addition and subtraction, and of multiplication and division. For simplicity, products and ratios are implemented using qualitative proportionalities;

```
;;; Abbreviations for signs
(=TR=> (Positive ?Q)       ( Pos ?Q))     (=TR=> (Increasing ?Q)       ( Inc ?Q))
(=TR=> (Non-Positive ?Q)   (~Pos ?Q))     (=TR=> (Non-Increasing ?Q)   (~Inc ?Q))
(=TR=> (Negative ?Q)       ( Neg ?Q))     (=TR=> (Decreasing ?Q)       ( Dec ?Q))
(=TR=> (Non-Negative ?Q)   (~Neg ?Q))     (=TR=> (Non-Decreasing ?Q)   (~Dec ?Q))
(=TR=> (Zero ?Q)           ( Zip ?Q))     (=TR=> (Constant ?Q)         ( Con ?Q))
(=TR=> (Non-Zero ?Q)       (~Zip ?Q))     (=TR=> (Non-Constant ?Q)     (~Con ?Q))

;;; Conversion of signs to inequalities
(=TR=> ( Pos ?Q)  (>  ?Q 0))     (=TR=> ( Inc ?Q)  (>  (D ?Q) 0))
(=TR=> (~Pos ?Q)  (<= ?Q 0))     (=TR=> (~Inc ?Q)  (<= (D ?Q) 0))
(=TR=> ( Neg ?Q)  (<  ?Q 0))     (=TR=> ( Dec ?Q)  (<  (D ?Q) 0))
(=TR=> (~Neg ?Q)  (>= ?Q 0))     (=TR=> (~Dec ?Q)  (>= (D ?Q) 0))
(=TR=> ( Zip ?Q)  ( = ?Q 0))     (=TR=> ( Con ?Q)  ( = (D ?Q) 0))
(=TR=> (~Zip ?Q)  (!= ?Q 0))     (=TR=> (~Con ?Q)  (!= (D ?Q) 0))

;;; Conversion to  >=
(=TR=> ( = ?N1 ?N2)  (and (>= ?N1 ?N2) (<= ?N1 ?N2)))
(=TR=> (!= ?N1 ?N2)  (not ( = ?N1 ?N2)))
(=TR=> (<  ?N1 ?N2)  (not (>= ?N1 ?N2)))
(=TR=> (>  ?N1 ?N2)  (not (<= ?N1 ?N2)))
(=TR=> (<= ?N1 ?N2)       (>= ?N2 ?N1))

;;; Anti-Symmetry
(==> (> ?N1 ?N2)  (not (> ?N2 ?N1)))

;;; Installing referenced quantities
(==> (and (>= ?N1 ?N2) (:TEST (not (derivative? ?N1))))
     (:ASSERT (Quantity ?N1)))

;;; QPE support
(=TR=> (Greater-Than ?N1 ?N2)  (> ?N1 ?N2))
(=TR=> (Less-Than ?N1 ?N2)     (< ?N1 ?N2))
(=TR=> (Equal-To ?N1 ?N2)      (= ?N1 ?N2))
```

Figure 3.4: Rules for Inequalities

```
;;; Same_Relations
(==> (Same_Rel (?N1a ?N1b) (?N2a ?N2b))
     (and (iff (> ?N1a ?N1b) (> ?N2a ?N2b))
          (iff (< ?N1a ?N1b) (< ?N2a ?N2b))))

;;; Sign relations
(=TR=> (Same_Sign ?N1 ?N2)  (Same_Rel (?N1 0) (?N2 0)))
(=TR=> (Opp_Sign  ?N1 ?N2)  (Same_Rel (?N1 0) (0 ?N2)))

;;; QPE support
(==> (and (Correspondence (?Q1 ?Q2) (?Q3 ?Q4)) (Qprop+ ?Q1 ?Q3))
     (Same_Rel (?Q1 ?Q2) (?Q3 ?Q4)))

(==> (and (Correspondence (?Q1 ?Q2) (?Q3 ?Q4)) (Qprop- ?Q1 ?Q3))
     (Same_Rel (?Q1 ?Q2) (?Q4 ?Q3)))

(=TR=> (Ordered-Correspondence ?P1 ?P2)  (Same_Rel ?P1 ?P2))
```

Figure 3.5: Rules for Related Ordinal Relations

```
;;; Sums
(==> (and (Affected ?Q +) (not (Affected ?Q -)))  (Pos ?Q))
(==> (and (Affected ?Q -) (not (Affected ?Q +)))  (Neg ?Q))
(==> (not (Affected ?Q +))  (~Pos ?Q))
(==> (not (Affected ?Q -))  (~Neg ?Q))

(==> (and (Sum_Member ?sum ?Q +) (Pos ?Q))  (Affected ?sum +))
(==> (and (Sum_Member ?sum ?Q +) (Neg ?Q))  (Affected ?sum -))
(==> (and (Sum_Member ?sum ?Q -) (Pos ?Q))  (Affected ?sum -))
(==> (and (Sum_Member ?sum ?Q -) (Neg ?Q))  (Affected ?sum +))

(=TR=> (Sum_Member ?sum ?Q)  (Sum_Member ?sum ?Q +))
(==> (Sum_Member ?sum ?Q +)  (Qprop+ ?sum ?Q))
(==> (Sum_Member ?sum ?Q -)  (Qprop- ?sum ?Q))
```

Figure 3.6: Rules for Sums

```
;;; Influences
(==> (and (Influenced ?Q +) (not (Influenced ?Q -)))  (Inc ?Q))
(==> (and (Influenced ?Q -) (not (Influenced ?Q +)))  (Dec ?Q))
(==> (not (Influenced ?Q +))  (~Inc ?Q))
(==> (not (Influenced ?Q -))  (~Dec ?Q))

;;; Indirect Influences
(==> (and (Qprop+ ?Q1 ?Q2) (Inc ?Q2))  (Influenced ?Q1 +))
(==> (and (Qprop+ ?Q1 ?Q2) (Dec ?Q2))  (Influenced ?Q1 -))
(==> (and (Qprop- ?Q1 ?Q2) (Inc ?Q2))  (Influenced ?Q1 -))
(==> (and (Qprop- ?Q1 ?Q2) (Dec ?Q2))  (Influenced ?Q1 +))

(=TR=> (Qprop   ?Q1 ?Q2)  (Qprop+  ?Q1 ?Q2))
(=TR=> (QpropO  ?Q1 ?Q2)  (QpropO+ ?Q1 ?Q2))
(=TR=> (QpropO+ ?Q1 ?Q2)  (and (Qprop+ ?Q1 ?Q2) (Same_Sign ?Q1 ?Q2)))
(=TR=> (QpropO- ?Q1 ?Q2)  (and (Qprop- ?Q1 ?Q2) ( Opp_Sign ?Q1 ?Q2)))

;;; Direct Influences
(==> (I+ ?Q ?Rate)  (Sum_Member (Net-Inf ?Q) ?Rate +))
(==> (I- ?Q ?Rate)  (Sum_Member (Net-Inf ?Q) ?Rate -))

(==> (or (I+ ?Q ?Rate) (I- ?Q ?Rate))  (:ASSERT (Deriv ?Q (Net-Inf ?Q))))

;;;; QSIM Support
(==> (Deriv ?Q1 ?Q2)      (Same_Sign (D ?Q1) ?Q2))
(==> (M+ ?Q1 ?Q2)         (Same_Sign (D ?Q1) (D ?Q2)))
(==> (M- ?Q1 ?Q2)         ( Opp_Sign (D ?Q1) (D ?Q2)))
(=TR=> (MO+ ?Q1 ?Q2)      (and (M+ ?Q1 ?Q2) (Same_Sign ?Q1 ?Q2)))
(=TR=> (MO- ?Q1 ?Q2)      (and (M- ?Q1 ?Q2) ( Opp_Sign ?Q1 ?Q2)))
```

Figure 3.7: Rules for Influences

however it is considered a modeling error for a product or ratio to be influenced outside of its definition.[3]

Figure 3.9 contains miscellaneous rules supporting various external and internal functionality. The first set of translations implement logical connectives. Translations are used to propagate negation to literals through the logical connectives and and or. The connective: `iff` translates into two disjunctive clauses, while `==>` (read "implies") translates into a single clause. Note that `==>` is used both as a logical connective and as the rule macro symbol. Internal implications within a rule consequent do not introduce new triggers, and do not support the rule keywords described in Section 3.2.3.2. The next two rules interpret correspondences from `QPE` models, using the `Same_Rel` predicate. The next three rules provide access to internal procedures given certain declarations in the model. The predicate `Fact-To-Close` signals a closed proposition; model closing is described in Section 3.5.1. The next rule uses the `When_Quantity`

[3]This policy constraint is not currently enforced.

```
;;; Algebra
(==> (and (:= ?Q1 ?Q2) (Quantity ?Q2))  (and (= ?Q1 ?Q2) (= (D ?Q1) (D ?Q2))))

;;;; Binary Addition and Subtraction
(==> (:= ?Qs (+ ?Q1 ?Q2))   (Sum_Rel1 ?Qs ?Q1 ?Q2))
(==> (:= ?Qd (- ?Q1 ?Q2))   (Sum_Rel1 ?Q1 ?Q2 ?Qd))

(=TR=> (Sum_Rel1 ?Qs ?Q1 ?Q2) (and (Sum_Rel ?Qs ?Q1 ?Q2) (Sum_Rel ?Qs ?Q2 ?Q1)))

(=TR=> (Sum_Rel ?Qs ?Q1 ?Q2)
     (and (Same_Rel (?Qs ?Q1) (?Q2 0))
          (Same_Rel ((D ?Qs) (D ?Q1)) ((D ?Q2) (D 0)))))

;;; Binary Multiplication and Division
(==> (:= ?Qp (* ?Q1 ?Q2))
     (and (Prod_Rel ?Qp ?Q1 ?Q2)
          (==> (Pos ?Q2) (Qprop+ ?Qp ?Q1))
          (==> (Neg ?Q2) (Qprop- ?Qp ?Q1))
          (==> (Pos ?Q1) (Qprop+ ?Qp ?Q2))
          (==> (Neg ?Q1) (Qprop- ?Qp ?Q2))))

(==> (:= ?Qr (/ ?Qn ?Qd))
     (and (Prod_Rel ?Qn ?Qr ?Qd)
          (==> (Pos ?Qd) (Qprop+ ?Qr ?Qn))
          (==> (Neg ?Qd) (Qprop- ?Qr ?Qn))
          (==> (Neg ?Qn) (Qprop+ ?Qr ?Qd))
          (==> (Pos ?Qn) (Qprop- ?Qr ?Qd))))

(=TR=> (Prod_Rel ?Qp ?Q1 ?Q2)
     (and (==> (Zip ?Q1)  (Zip ?Qp))
          (==> (Zip ?Q2)  (Zip ?Qp))
          (==> (and ( Pos ?Q1) ( Pos ?Q2))  ( Pos ?Qp))
          (==> (and ( Neg ?Q1) ( Neg ?Q2))  ( Pos ?Qp))
          (==> (and ( Pos ?Q1) ( Neg ?Q2))  ( Neg ?Qp))
          (==> (and ( Neg ?Q1) ( Pos ?Q2))  ( Neg ?Qp))
          (==> (and (~Pos ?Q1) (~Pos ?Q2))  (~Neg ?Qp))
          (==> (and (~Neg ?Q1) (~Neg ?Q2))  (~Neg ?Qp))
          (==> (and (~Pos ?Q1) (~Neg ?Q2))  (~Pos ?Qp))
          (==> (and (~Neg ?Q1) (~Pos ?Q2))  (~Pos ?Qp))))
```

Figure 3.8: Rules for Algebraic Operators

```
;;; Handling connectives
(=TR=> (and ?P)  ?P)      (=TR=> (or ?P)  ?P)      (=TR=> (not (not ?P))  ?P)
(=TR=> (not (and ?P1 ?P2 . ?rest))  (or  (not ?P1) (not (and ?P2 . ?rest))))
(=TR=> (not (or  ?P1 ?P2 . ?rest))  (and (not ?P1) (not (or  ?P2 . ?rest))))
(=TR=> (==> ?P1 ?P2)      (or ?P2 (not ?P1)))
(=TR=> (if ?P1 ?P2)       (==> ?P1 ?P2))
(=TR=> (if ?P1 ?P2 ?P3)   (and (==> ?P1 ?P2) (==> (not ?P1) ?P3)))
(=TR=> (iff ?P1 ?P2)      (and (==> ?P1 ?P2) (==> ?P2 ?P1)))
(=TR=> (xor ?P1 ?P2)      (not (iff ?P1 ?P2)))

;;; QPE support
(=TR=> (A ?Q) ?Q)
(=TR=> (Q= ?Q ?Form)  (:= ?Q ?Form))

;;; Hooks from declarations to procedural handlers
(==> (Function_Spec . ?args)  (:EVAL (iqe:process-function-spec ?args)))

(==> (Fact_To_Close ?X)  (:EVAL (fact-to-close ?X)))

(==> (and (When_Quantity ?Q . ?facts) (Quantity ?Q))  (and . ?facts))

(==> (or (M+ ?Q ?X) (M- ?Q ?X) (Deriv ?Q ?X) (:= ?Q ?X))  (Defined ?Q))
```

Figure 3.9: Miscellaneous Rules

predicate to conditionally introduce certain facts about a quantity only when the quantity is already known to the model; this provides fine control for selecting only those parts of a domain theory which are required for solving a particular problem. Finally, the last rule identifies `Defined` quantities whose derivatives are determined as a function of one or two other quantities; this bypasses the costly mechanism for resolving influences required for qualitative proportionalities.

3.5 Closed Models and Default Reasoning

Default reasoning, unlike ordinary logic, allows inferences of the form: "Believe P, unless there is reason to believe $\neg P$". Default reasoning captures what is typical, normal or expected. For example, it is reasonable to assume that your car is where you parked it, unless you have reason to believe that it was towed, stolen, or picked up by your spouse. As does most of default reasoning, this example concerns the classic frame problem [55] and the notion of persistence. Default reasoning in general allows reasonable assumptions to be made given incomplete information.

Default reasoning is by its nature nonmonotonic. In ordinary monotonic logic, adding a proposition results in a larger set of valid inferences; if a proposition is believed before the addition, it will continue to be believed after the new proposition is added. In default reasoning, a default belief may be invalidated by a newly-added piece of information; the default belief

```
(I+ <q> <q>)   (Qprop+ <q> <q>)
(I- <q> <q>)   (Qprop- <q> <q>)

(Sum-Member <q> <q> <s>)

(Affected <q> <s>)    (Influenced <q> <s>)

(:= <q> (<op> <q> <q>))
```

Figure 3.10: Summary of Domain-Independent Closed Propositions

and all of its consequences must be retracted from the set of beliefs after adding the conflicting proposition.

Default reasoning is used extensively in both model-based diagnosis and qualitative reasoning. In model-based diagnosis, the model is initially assumed to be correct; that is, its components are assumed to be working correctly by default. The possibility of a failure is only considered when the preferred default beliefs of correct operation are inconsistent with observations. Diagnosis is clearly a form of default reasoning. The best (most plausible) diagnosis is the one which violates as few default beliefs as possible, taking into account the relative strengths of each default belief.

In the process-centered models of QP theory, entities are assumed not to exist unless the model specifically sais they exist. A quantity is assumed to be uninfluenced (and therefore constant) by default, and is only influenced to the extent predicted by the model. That is, the model is assumed to be *closed* with respect to influences on quantities. This section discusses the nature of the closed models of QP theory, emphasizing their importance in model-based diagnosis.

3.5.1 Model Closing

The ability to dynamically compose a model for a complex system from a domain theory and a scenario description is one of the greatest benefits of the process-centered approach. The actual behavior resulting from a given process can only be determined in the context of the other processes (if any) influencing the same quantities. Thus most of a model's predictions require an additional assumption that the model is complete—that the processes known to the model are in fact the only processes acting to influence the quantities of interest. More precisely, the model is assumed to be *closed* with respect to active processes and the resulting influences on quantities. This assumption is necessary, for example, to infer that a quantity is uninfluenced and therefore constant. The set of domain-independent predicates which must be closed is summarized in Figure 3.10.

A domain theory specifies (among other things) under what conditions each potential influence on a quantity is in effect. Because a domain theory must allow for an open-ended combination of processes influencing a quantity, it cannot say under what conditions a quantity is *not* influenced. Only at run-time is it possible to identify the set of known potential influences on a quantity, and determine the conditions under which none are in effect. This procedure is called *closing* the model, and can only be performed at particular points in the inference process, when specific kinds of inferences have been completed.[4] Model closing is accomplished through the completion, or closing of individual propositions within the model, as described below.

3.5.1.1 Closing Propositions

In a non-Horn model, a proposition and its negation may both have consequences. Thus it is important to know not only when a proposition is true but also when it is false. A literal may default to either true or false; however in model closing a positive literal is typically considered false unless the model provides a reason to believe it is true. A proposition is defaulted to false through a procedure called *proposition closing.*

Because a closed proposition is false by default, it represents an exception to the norm. Closing an exceptional proposition is in effect saying that the *known* reasons for believing the exception are in fact the *only* reasons for believing it. If all the possible reasons for believing an exception are checked and eliminated, then the exception is (assumed to be) false.

Closing a proposition corresponds to the notion of "completion" introduced in [3]. Logically the completion of a literal m may be viewed as follows. Given a set of causes for m,
$c_1 \Longrightarrow m$
$\dots$
$c_n \Longrightarrow m$
we complete m by adding the inverse implication:
$m \Longrightarrow c_1 \vee c_2 \vee \cdots \vee c_n$
This sais in effect that m can only occur if one or more of its causes occurs. From this we may infer that m is false exactly when all of its causes are false.

The causes of a proposition are represented by its incoming justifications. A closed proposition should be disbelieved whenever all of its incoming justifications have at least one false antecedent. Thus a proposition is closed by justifying its negation with combinations of negated antecedents from its incoming justifications.

Definition 1 (Closed Propositions) *Let P be a proposition with incoming justifications $\mathcal{J}_P$. Proposition P is considered* closed *when $\neg P$ is justified by each combination of negated antecedents drawn one from each $J \in \mathcal{J}_P$.*

Closing proposition P involves installing the appropriate justifications for $\neg P$, based on the justifications $\mathcal{J}_P$ for P. In the simple case where $\mathcal{J}_P$ is empty, closing P makes $\neg P$ true (and P false). If $\mathcal{J}_P$ consists of a single justification having a single antecedent A, P is closed by adding the justification: $\neg A \Longrightarrow \neg P$. In general, $\neg P$ is justified whenever every J in $\mathcal{J}_P$ contains a false antecedent. Note that the justifications added for $\neg P$ while closing P are only valid if the known justifications for P are complete; discovering a new justification for P *violates* the closing. Violated proposition closings are discussed in Section 3.5.3.1.

Figure 3.11 shows an example of the justifications needed to close a proposition. Figure 3.11 a) shows the existing justifications on proposition P_d, and Figure 3.11 b) shows the justifications installed on $\neg P_d$ to close P_d. For each combination of antecedents chosen one from each

[4] Note that this need to delay closing creates an exception to the requirement that all knowledge be explicitly represented as rules. This is discussed further in Chapter 5.

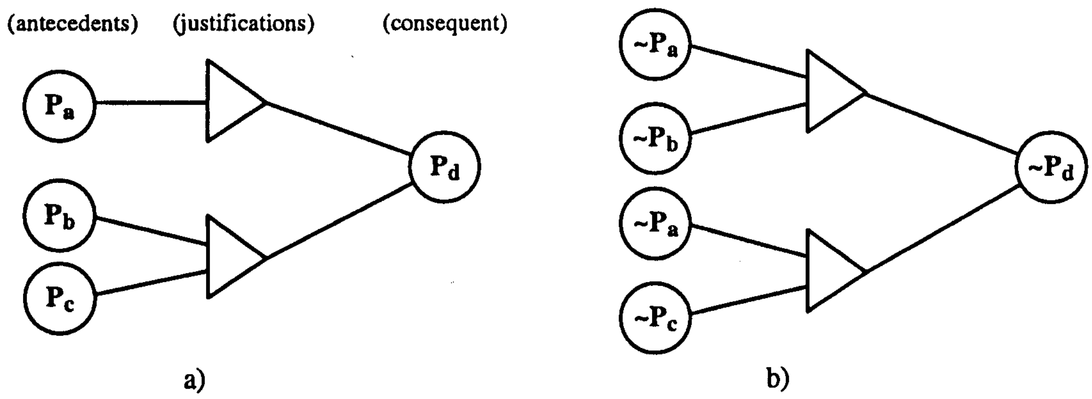


Figure 3.11: Justifications Needed for Closing a Proposition P_d

justification for P_d, a new justification for $\neg P_d$ is constructed. The antecedents for the new justification are the negations of the chosen antecedents. The effect of closing a proposition is to guarantee that its truth status will be known regardless of the truth or falsity of its antecedents. In other words, a closed proposition has a unique truth assignment given any truth assignment for the antecedents of its incoming justifications:

Theorem 1 (Closure Completeness) *Let P be a closed proposition with incoming justifications $\mathcal{J}_P$. Let $\mathcal{A}_P$ be the antecedents contained in $\mathcal{J}_P$. For any truth assignment on $\mathcal{A}_P$, P has a unique truth assignment.*

Proof: Either some justification on P has all true antecedents, in which case P is true, or every justification has at least one false antecedent. We must show that in the latter case P is false. Pick one false antecedent from each justification; call the resulting set $\mathcal{A}_f$. P is closed, so there must be a justification on $\neg P$ for every combination of antecedents chosen one from each justification on P. In particular, there must be a justification on $\neg P$ whose antecedents are the negations of the propositions in $\mathcal{A}_f$; call this J_f. Because every element in $\mathcal{A}_f$ is false, every antecedent of J_f must be true. Therefore J_f succeeds in justifying $\neg P$, making P false. □

It is straightforward to automate the process of closing a proposition. The proposition: `(Fact-to-Close` *<prop>*`)` identifies *<prop>* as a proposition to be closed. In addition, the form: `(defClosed-Predicate` *<pred>*`)` indicates that all propositions of the form: `(`*<pred>* `.` *<args>*`)` are to be closed. Automatic proposition closing requires building the appropriate justifications on the negated proposition. Useless justifications are avoided by ignoring true antecedents in the original justifications. Also, justifications containing false antecedents may be ignored.

3.5.2 Closing Arbitrary Sets

Closing propositions may be viewed as a special case of a general problem of reasoning about sets. When closing a proposition, the set of incoming justifications is closed. In general, any

```
(==> (and (Set-Predicate ?set ?pred) (Set-Member ?set ?member))
     (and (==> (?pred ?member)  (Some ?set ?pred))
          (==> (not (?pred ?member))  (not (Every ?set ?pred)))))

(==> (Set-Predicate ?set ?pred)  (:EVAL (index-set-pred ?set ?pred)))
```

Figure 3.12: Rules for Closing Sets

arbitrary set of objects may be closed, thus enforcing that the known members of the set are the only members.

Consider a general query of the form: "Does any member of set S exhibit property P?"; or stated formally: $\exists s \in S : P(s)$. One way to answer this query in the affirmative requires finding an existence proof—some s in S exhibiting $P(s)$. On the other hand, proving that no such s exists in S requires showing that for every s in S, s does not exhibit property P; that is: $\forall s \in S : \neg P(s)$. In the kind of reasoning we are doing here, such queries require knowing the complete extension of the set, in addition to knowing of the absence of the property for each (known) member of the set.

Figure 3.12 contains the rules necessary for implementing sets. Sets are defined incrementally—that is, one member at a time—using the `Set-Member` predicate. Predicates of interest to the set are declared using `Set-Predicate`. The two predicates: `Some` and `Every` are maintained for each set predicate. Inferring that some member (or not every member) of a set exhibits some property is straightforward, as the first rule shows. Inferring that every member (or no member) exhibits some property requires closing the set procedurally, which is handled by the second rule. Closing a set predicate uses the same mechanism as is used for closing an ordinary proposition. That is, propositions using the predicate `Some` are closed with a default of false, while propositions containing `Every` are closed with a default of true. Note that these defaults match the base behavior of "some" and "every" as applied to an empty set.

3.5.3 Volatile and Nonvolatile Closing

Automated proposition closing can be used in different ways. In some cases, it is used as a modeling shorthand. When all the justifications for a given proposition are known during domain theory construction, the model builder has enough information to manually close the proposition, by justifying its negation appropriately. Tagging the proposition for closure eliminates the need to manually install the negative justifications.

The real benefits of automatic proposition closing occur when the eventual number of justifications for the proposition depends on the particular scenario, and so cannot be known during domain theory construction. Proposition closing is essential for compositional models, as it allows the membership of a set to be established dynamically at run time, after the entire scenario description has been processed.

As mentioned previously, the justifications added for $\neg P$ while closing a proposition P are only valid when all incoming justifications for P are known in advance; if a new justification for

P is discovered, then P's closing is *violated*, and all of the closing justifications for $\neg P$ become invalid.

Propositions closed as a modeling shorthand express a constraint of the domain theory, and are generally independent of any particular modeled system. Because diagnosis does not typically bring into question the constraints of the domain theory, this type of closing is considered *nonvolatile* within the diagnostic task.[5] The distinction between volatile and nonvolatile proposition closing is important because it allows us to avoid the overhead associated with the reclosing mechanism discussed in the next section.

In diagnosis, failures may introduce new structure which in turn may introduce new justifications on closed propositions. Proposition closings which depend on the specifics of the scenario description must be viewed in diagnosis as *volatile*, so that the possibility of hidden processes may be handled.

Certain closings within the domain theory must be considered volatile, even though the domain theory itself is not questioned. In particular, if the antecedents of a rule contain additional variables not contained in the consequent, then the closing of that consequent must be considered as volatile. This is true because the number of justifications for a given propositional consequent (i.e., a given set of bindings) is dependent on the particular scenario. Such cases are detected automatically during proposition closing, by examining the rules which could introduce justifications for the proposition being closed. Any rule whose consequent unifies with the proposition, and which has not already instantiated in all possible ways,[6] causes the closing to be marked as volatile. The next section discusses the handling of volatile proposition closings.

3.5.3.1 Closed-World Assumptions (CWAs)

As stated previously, most of a model's predictions require the assumption that the processes known to the model are in fact the only processes acting to influence the quantities of interest. That is, we must assume that the model is closed with respect to the sets of influences on quantities. Having made this *closed-world assumption*, we may, for example infer that an uninfluenced quantity must be constant.

The explicit representation of closed-world assumptions provides a means of updating beliefs in light of new information. For example, if a quantity originally has no known influences but is later discovered to be influenced positively (or negatively), then the original closed-world assumption was incorrect. The old closed-world assumption must be retracted (i.e., removed from the set of current beliefs), and the proposition must be reclosed using a new closed-world assumption encompassing the newly-discovered influence.

Definition 2 *Closed-World Assumptions Suppose P is a volatile closed proposition. Associated with P is a closed-world assumption representing the currently known justifications for P. The complementary functions cwa() and closed_prop() map one-to-one from volatile closed propositions to their closed-world assumptions and back.*

By (implicit) assumption, a nonvolatile closed proposition will never have additional justifications, and so does not require an explicit closed-world assumption. Each volatile closed propo-

[5]If we allowed for modeling errors within the domain theory, as theory revision does, then these closings would have to be brought into question.

[6]This includes rules which have not triggered at all, even if all of their variables are used in the consequent.

sition P has an associated closed-world assumption $cwa(P)$, representing the belief that the known justifications for P are in fact the only ones. Viewed differently, the negation of $cwa(P)$ implies that another justification for P exists. This can be represented as: $\neg cwa(P) \wedge X \Longrightarrow P$, for some unknown X.[7] If we explicitly install this justification before closing P, all of the closing justifications on $\neg P$ will contain one additional antecedent; namely, either $cwa(P)$ or $\neg X$. Because X represents an unknown, we will never know when it is false, so there is no use in constructing justifications containing $\neg X$. Thus all the closing justifications contain $cwa(P)$ as an antecedent, and the consequences of the closing will follow only as long as the closed-world assumption is believed. Closed-world assumptions can be viewed as marking the points within the set of propositions (and their justifications) where additional structure could make a difference.

3.5.3.2 Reclosing Propositions

Closed propositions are linked to their corresponding closed-world assumption, in order to handle closed-world assumption violations. The addition of a new justification on a volatile closed proposition constitutes a closed-world assumption violation and automatically invokes a reclosing mechanism.[8] The proposition is reclosed by first retracting the associated closed-world assumption from the set of default beliefs, and then closing the proposition as before. This results in a new closed-world assumption which takes into account the new justification. This process may be repeated many times for the same closed proposition as more justifications are discovered.

Note that the new justification may turn out to have a false antecedent, rendering it useless. In such a case the old and the new closed-world assumptions are logically equivalent. The new closed-world assumption does not require retraction, and there is no need to backtrack to the old closed-world assumption. For this reason, violated closed-world assumptions are "garbage collected" by making them false, thereby preventing their further consideration.

3.5.3.3 Efficient Closing

Consider the justification structure in Figure 3.13. Proposition P has several justifications, each with multiple antecedents. The number of new justifications required to close P is equal to the product of the sizes of the justifications for P. In cases where the justification structure is highly connected, this number can become prohibitively large, with a corresponding degradation in run-time performance.

One (suboptimal) solution to this problem is to rewrite the domain theory and the domain-independent rules so as to modify the resulting justification structure. Intermediate nodes may be introduced—one for each conjunctive justification, as shown in Figure 3.14. This breaks up the combinatorics by providing a proposition representing the satisfiability of each justification's antecedents. The resulting number of justifications required for closing is reduced to (one greater than) the total number of antecedents in the original justifications.

[7]Negated closed-world assumptions correspond roughly to the "anonymous causes" introduced by Console and Torasso [8].

[8]Violating a non-volatile closed proposition constitutes a hard error; only volatile closings can be undone.

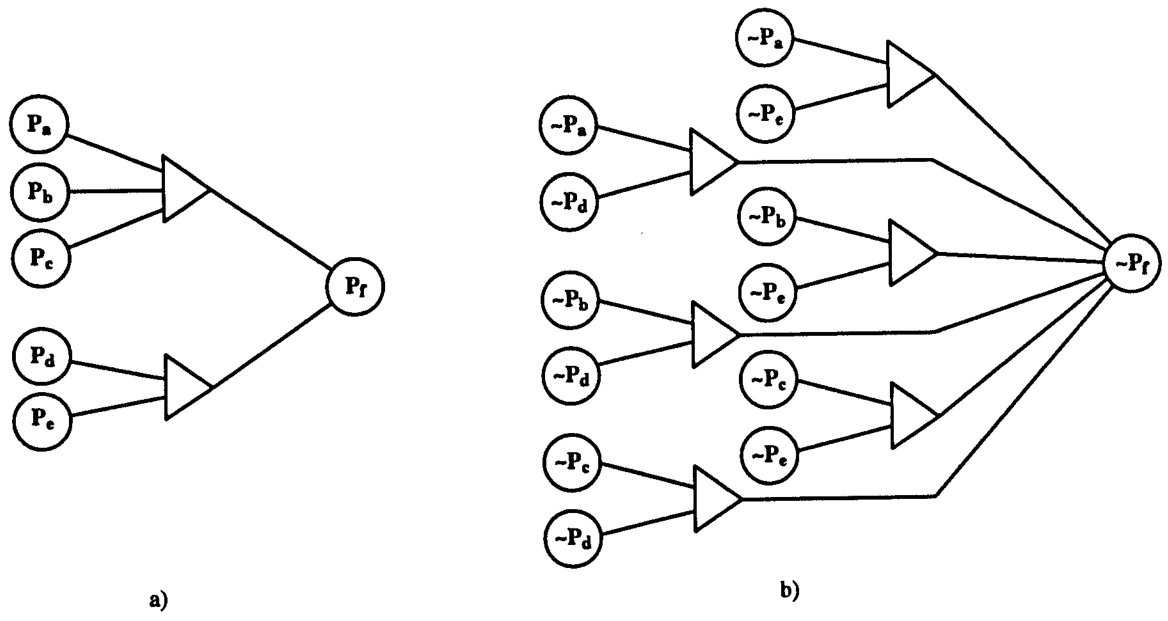


Figure 3.13: Combinatorics of Simple Proposition Closing

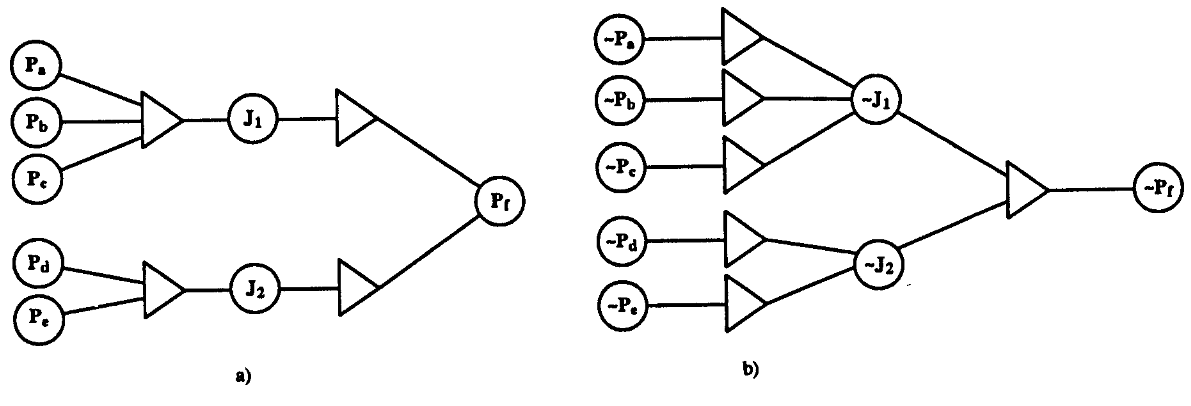


Figure 3.14: Efficient Proposition Closing using Intermediate Propositions

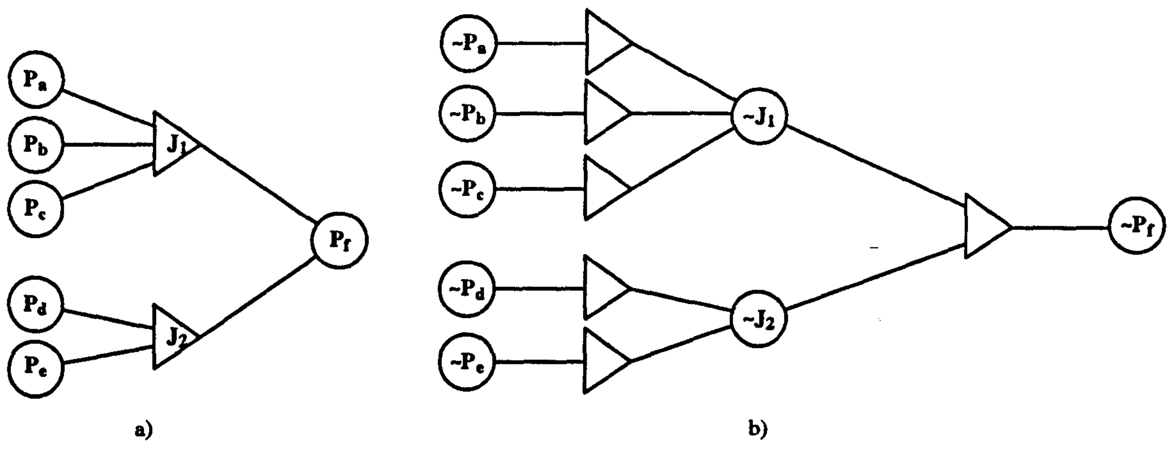


Figure 3.15: Automatic Creation of Intermediates During Proposition Closing

The added efficiency of this approach comes at the expense of simplicity of the model. Domain theories require more time to construct, and become more difficult to understand. In addition, it is unreasonable to expect the model builder to be aware of such efficiency issues.

The solution taken here is to automate the mechanism for introducing intermediate propositions, as shown in Figure 3.15. These *hidden* propositions are only needed for the closing justifications on the default proposition; the original justifications are unaffected. The number of hidden propositions introduced is equal to the number of original justifications having more than one antecedent. Each receives as many justifications as its corresponding original justification has antecedents. Each new justification contains a single antecedent; thus no cross-products are required, making their labels very inexpensive to maintain. One additional justification links the hidden propositions to the negation of the proposition being closed. Empirical results have shown that the reduced combinatorics of this approach far outways the minimal overhead required to construct the intermediate nodes and maintain their labels.

3.5.4 Relation to Default Reasoning

Model closing is a form of default reasoning. A closed proposition is false by default, and is believed only when justified by the model. With the inclusion of explicit closed-world assumptions and the supporting mechanisms for reclosing violations, a closed model is nonmonotonic; adding a new piece of information can lead to a new justification on a closed proposition, causing it to be reclosed and the consequences of the original closing to be in effect retracted.[9]

However, model closing differs from most other default reasoning in one important respect. Just as a building is only as strong as its foundation, a closed proposition is only as complete as its antecedents. A closed proposition is only guaranteed a truth assignment when all of the antecedents of its incoming justifications have truth assignments. Unknown truth assignments can propagate through a closed model. Thus for the closing to be effective, the model builder

[9]The underlying ATMS used in this research is monotonic; it is the manipulation of the set of default beliefs which yields nonmonotonic behavior.

must also close the antecedents of the incoming justifications for a closed proposition, all the way back to the ground assumptions underlying the model.

The model must specify which propositions default to true and which default to false. Entities, views and processes from the domain theory are automatically closed with a default of false—objects do not exist unless the model says they do. Any conditions on views and processes which are not closed automatically must either be ground assumptions or closed manually by the model builder. Some propositions—inequalities for example—do not have a meaningful default value and so are not closed; this is one explanation for why quantity conditions must be assumed.

3.5.5 Diagnosing Violated Closed-World Assumptions

As described above, closed-world assumptions are introduced to represent the belief that the known members of some set are in fact the only members. A closed-world assumption can become violated if a failure has the effect of introducing new members to a closed set. Primarily the sets of interest are the sets of influences on quantities. Each process influencing a given quantity contributes one member to the set of influences for that quantity.

Set membership is generally only part-time. Processes come and go as their conditions are met and unmet. Each potential member knows when it is a member and when it is not. In particular, each member is responsible for removing itself from the set should that become necessary. The closed-world assumption does not dictate that all potential members of a set are in fact members; it merely states that there are no other members. Thus removing a potential member from a set does not violate the closed-world assumption; this only occurs when an unanticipated entity joins the known members of the set.

3.5.5.1 Failures of Omission

As stated in Chapter 2, there are two basic types of failures in any model: failures of *commission*, and failures of *omission*. These correspond to violations of the logical properties of soundness and completeness, respectively. A failure of commission is unsound; a failure of omission is incomplete. Closed-world assumption violations result from an incompleteness of the model. Either the scenario description is missing some structural detail which would explain a missing process, or the domain theory is missing a process type or is overly restrictive on the conditions of a known process type. The explicit representation of closed-world assumptions provides a way of identifying and handling such failures of omission.

As an example of how a closed-world assumption can be violated, consider a leak in the bottom of a vat of liquid. There may be several known processes moving liquid into and out of the vat; but the leak was unanticipated, and thus was not included in the set of possible negative influences on the amount of liquid in the vat. The corresponding closed-world assumption has been violated, requiring a new closed-world assumption to be made which includes the newly-discovered leak.

Identifying closed-world assumption violations is in principal no different from identifying component failures. Closed-world assumptions represent default beliefs, which may be retracted if necessary to eliminate an inconsistency. By explicitly representing closed-world assumptions and including them in the set of defeasible assumptions, the same algorithm (presented in Chapter 4) which finds violated component correctness assumptions will find violated closed-world assumptions as well.

Closed-world assumption violations do not constitute a diagnostic candidate in the traditional sense. They do not describe physical structure explicitly, and do not precisely identify which component has failed. The negation of a closed-world assumption has no consequences, so embracing the negation provides no additional predictions for the model. A candidate involving a closed-world assumption violation is viewed as an intermediate result, which requires further explanation.

The desired explanation for a closed-world assumption violation takes the form of a structural description, specifying what new entities exist and how they interconnect with the rest of the system. This augmented scenario description should indirectly specify the type of process(es) responsible for the closed-world assumption violation, including the entities involved and the roles they play. Because of the combinatorics of the different ways in which this can occur, it is not feasible to have enumerated these possibilities in advance. Thus there is no way to identify these using a consistency-based approach.

The approach taken here to explain closed-world assumption violations is to perform a backward chaining search through the domain theory for a set of structural entities and connections explaining the violation. Chapter 5 describes an abductive backchaining algorithm which uses ATMS label propagation to implement this search, and discusses specifically its application to explaining closed-world assumption violations.

3.5.5.2 Failures of Commission

It is also possible for a model to fail due to unsoundness of either the scenario description or the domain model. For example, a clogged pipe does not constitute a fluid path, so that part of a scenario description is unsound. A process definition may be missing conditions, and thus incorrectly predict activity of a process instance. Repairing such model failures requires retracting an assumption representing the soundness of the domain theory or the scenario description (as described in the next section), and does not involve the proposition reclosing mechanism discussed above.

Model failures may thus be categorized into one of four types: an unsoundness or incompleteness of either the domain theory or the scenario description. Because this research is about diagnosis and not theory revision, we must assume that the domain theory is correct—that is, both sound and complete. Incompleteness of the scenario description is (at least partially) handled by proposition reclosing and retraction of violated closed-world assumptions. This leaves the unsoundness of the scenario description. We require the ability to retract individual beliefs about the structure of a system. This requires that each proposition of the scenario description be assumed rather than asserted. These assumptions along with several other types of defeasible assumptions considered in this research are discussed in the next section.

3.6 Defeasible Assumptions

One of the ways this research differs from traditional model-based diagnosis is in its expansion of the set of *defeasible assumptions*—those aspects of the model which we are willing to bring into question. Other researchers in model-based diagnosis have limited their focus to component assumptions, indicating whether a component is working correctly or is in some other (unknown) mode. In this research we explore the nature of a failure by searching for a complete structural description for the device after failure.

When a component of a certain type fails, it effectively stops behaving as a component of that type, and possibly begins behaving as a component (or component assembly) of some other type. Some failures effectively eliminate certain components, while other failures introduce new structural entities which interact with the known structure to produce the observed behavior. The goal of diagnosis is to make the corresponding changes to the model—to bring it back into agreement with the actual physical structure.

Diagnosis is a search for a structural description of a failed system—not just in terms of which components have failed, or which of several pre-identified failure modes have occurred, but in terms of the structure which exists after failure. The explicit representation of closed-world assumptions introduced in the previous section provides the mechanism for positing new structure resulting from a failure. This section identifies several other types of *defeasible assumptions* which together provide the ability to search the space of the domain vocabulary for a consistent model of the failed system. In addition to the closed-world assumptions already discussed, PDE manipulates *structural assumptions*, *mode assumptions*, *modeling assumptions*, *domain theory assumptions*, *drift assumptions* and *observation assumptions*. Each of these is described in turn.

3.6.1 Structural Assumptions

Every component in any real system can fail. Some failures introduce spurious behaviors into the system, while others have the effect of blocking or otherwise preventing expected behaviors. In certain cases, a component failure causes the system to behave exactly as if that component were physically removed from the system. For example, an open resistor behaves identically to no resistor at all. Such failures are referred to as *nullification failures*, and the component is said to be *nullified* by the failure.

Nullification failures occur among assemblies as well as components. For example, if an automobile's ignition system fails to deliver a spark, it is effectively nullified as far as the rest of the engine is concerned. Similarly, a leak in a brake line can lead to loss of brake function, commonly described as "no brakes"; of course, the brakes themselves and most of the brake system components are still physically present, but you wouldn't know it from hitting the brake pedal.

Given a nullification failure, the model may be repaired by removing the component assumption from the set of beliefs, and replacing it with its negation. Alternatively, a component may be effectively removed from the model by retracting and negating the connectivity assumption(s) which connect it with the rest of the system. Each repair has the same effect; which one is more likely depends on the domain and the type of component involved. Some devices, such as a plug-in Christmas tree light, may be relatively likely to become disconnected, while a screw-in bulb may be more likely to fail outright. Also, in a domain such as electronics, simple disconnection of a wire or other component may occur without side-effects, whereas in a fluid system the physical disconnection of a pipe may introduce other possible paths for flow. Both connectivity and component assumptions provide the desired ability to localize a failure; however, they may suggest very different repair actions. Distinguishing between them is only possible by physically checking the connection and/or the component. To avoid generating both types of candidates, the model builder may choose connectivity assumptions as having a much higher probability than component assumptions (or vice versa), so that the more likely assumptions are brought into question only after exhausting other possibilities.

3.6.2 Mode Assumptions

One of the primary goals of this research is the ability to diagnose unanticipated failure modes. However, this does not preclude the use of available knowledge concerning common modes of failure. Common faults may be explicitly modeled as alternate modes of behavior for a component.

One might ask, "Why bother with explicit fault models when every failure is explainable in terms of violated component and/or closed-world assumptions?". For example, an open resistor represents a nullification failure, because it behaves as if the resistor had been removed from the circuit. This mode of failure is so common among resistors that it is natural to include a fault model for it. The benefit is that the fault model—conditioned by a *fault mode assumption*—provides a convenient hook for including any failure rate information for this common failure mode. Probabilities on component and closed-world assumptions cannot capture the likelihood of every common failure mode. Consider a shorted capacitor, which in effect introduces a wire across its terminals. This failure is clearly representable in terms of other components (a wire), and may be diagnosed as a violated closed-world assumption. But this closed-world assumption violation also represents an infinity of other possible instantiations, so a single probability on the closed-world assumption cannot capture the frequency of this particular failure mode. This requires an explicit fault model.

Components having common, frequently-occurring failure modes may include explicit fault models represented as behavioral modes. For example, a model for an electrical fuse could include a mode for "blown", modeled as an open circuit. Some researchers [27, 75] have extended this approach by adding an additional *unknown mode* to the set of possible behavioral modes. The unknown mode has no consequences; its purpose is to provide a way out of the constraint that a component must be in one of its known modes. This is meaningful only when probabilities or some other ordering criteria are used to favor the known modes over the unknown mode. The use of probabilities on assumptions is discussed in Chapter 4.

Mode assumptions are also needed to represent the operating region of some components which have not failed. Certain types of components have multiple operating regions which must be distinguished and modeled separately. The mode of a component is typically determined as a function of its inputs; for example, a transistor is in one of the three modes: {off, linear, saturated}, depending on its base-to-emitter voltage. Components having internal state—a flip flop, for example—can have modes which depend additionally on their state.

Components whose model depends on internal state or the values of its inputs may be in a different operating region than believed, and still be working "correctly" from the perspective of the model. For example, a valve which was believed to be open may in fact be closed, explaining why there is no flow through it. A digital counter may contain a high count due to receiving spurious inputs since its last reset. By having one mode assumption associated with each qualitatively interesting operating region for a component, we can represent our current beliefs regarding its status, and consider retracting those beliefs in order to explain a discrepancy.

Behavioral modes are represented by special modeling assumptions called *mode assumptions.* Each behavioral mode has an associated model fragment defining the behavior of that component while in that mode. The mode assumption becomes an additional antecedent in any inference sanctioned by the model fragment. Each component must be in exactly one of its known modes at any given time; that is, exactly one mode assumption must be true for each component.

It is useful to distinguish two types of modes: those which are determined within the model and those which are external to the model. An example of an external mode is a manually operated valve. The model does not specify how or when the valve changes from open to closed or vice versa; that is up to the whims of a human operator. An example of an internal mode is a float in a container of liquid. The float is floating whenever the level of liquid exceeds a certain height, and is not floating otherwise. The status of internal modes is usually defined by one or more inequalities, which in QP theory are the quantity conditions of views and processes. Quantity conditions are already represented as assumptions, so no other mode assumption is required for them. External modes are usually modeled as preconditions of views and processes, and are also assumed. Thus mode assumptions correspond to the normal state-distinguishing assumptions in `QPE` and `IQE`.

In some cases the mode of a component is completely determined by the current state of its upstream components; this is true for components lacking internal state. For example, a short circuit somewhere upstream of a transistor may force it to change from the *linear* mode to the *off* mode. When we examine the candidate corresponding to this short, we will be forced to change our beliefs about the status of the transistor as well. Derivable assumptions may be used in some cases to reason hierarchically about failures; this is discussed in Section 4.4.3.2.

3.6.3 Modeling Assumptions

In some cases, the model may yield inconsistent predictions because we are thinking about the system in the wrong way. A change in perspective or a less idealistic view may be sufficient to repair the model. Modeling assumptions are included in the domain theory to conditionally enable portions of the theory. Modeling assumptions allow a single domain theory to contain alternate models for a component or assembly; these may be individually selected at time of instantiation, or may coexist to support simultaneous consideration of mutually inconsistent perspectives.

Modeling assumptions may be used in a variety of ways. Some modeling assumptions specify a choice between two or more mutually incompatible models for a component or process. This choice may be between different levels of detail, different time scales, or different ontological commitments. For some apparent "failures", switching to an alternative model may cause the discrepancy to disappear.

Another use of modeling assumptions is to control the range of behaviors to be considered. For example, a steady state assumption rules out all behaviors involving change. In general, a modeling assumption of the form `(Disallow` *<behavior>*`)` rules out all behaviors containing *<behavior>*. If the observed symptoms include a disallowed behavior, then naturally the first repair of the model should be to allow it to consider the observed behavior. In general a disallowed behavior may be occurring but not be directly observed; thus these modeling assumptions should be included in the set of defeasible assumptions $\mathcal{D}$.

Modeling assumptions may also be used to selectively enable portions of the theory which may only be relevant in certain cases. Examples include simplifying assumptions of the form `(Ignore` *<quantity>*`)`, where *<quantity>* is some negligible parameter such as the friction or inertia of a pulley. In general, any parameter may be considered negligible, meaning its value is to be treated as zero. The current implementation requires that these be specifically modeled within the theory. For efficiency reasons, the simplified model usually does not even mention the negligible quantity; a smarter compiler could automatically build these simplified models for each potentially negligible quantity. If a failure causes an ignored quantity to become significant (e.g., a sticking pulley), then retraction of that simplifying assumption may repair the model.

Including modeling assumptions in the set of defeasible assumptions comes close to the border between diagnosis and theory revision. Given the generality of this approach, it is a natural extension to take the plunge, as the next section shows.

3.6.4 Domain Theory Assumptions

The techniques for finding failed assumptions presented in Chapter 4 do not care about the origin or nature of the assumptions. Existing diagnostic techniques generalize naturally to debugging domain theories. This requires including *domain theory assumptions* as additional antecedents for each culpable inference sanctioned by the domain theory.

For example, suppose the domain theory specifies that a flow process is active whenever a specific set of conditions is satisfied. Every instance of this flow process results in a justification whose antecedents are the conditions plus a domain theory assumption, and whose consequent represents the active flow process instance. If the theory omits some requirement from the conditions of the process, and erroneously predicts that the flow will occur when that requirement is unsatisfied, then the domain theory assumption may be held accountable. If on the other hand the theory includes some extraneous requirement in the conditions, then it will incorrectly predict inactivity when that requirement is unsatisfied. This case is detected as a closed-world assumption violation,[10] given the new justification (i.e., a generalization of the original one) for believing the process is active. By looking for sets of assumptions whose retraction is consistent with observations, the set of possible errors in the domain theory may be narrowed down to a manageable size.

Domain theory assumptions have proven useful in developing and debugging domain theories, by suggesting possible explanations for unexpected conflicts. Because this research focuses on diagnosis rather than theory revision, domain theory assumptions are not normally enabled during diagnostic problem solving. It is always possible to blame any discrepancy on some aspect of the domain theory, and the large number of assumptions involved makes their inclusion somewhat costly. Excluding domain theory assumptions is in effect making an implicit assumption that the domain theory is sound. Likewise, non-volatile proposition closings make an implicit assumption that the domain theory is complete. If performance is not critical, domain theory assumptions (and additional closed-world assumptions) may be included to represent the soundness (and completeness) of the domain theory; these should be given suitably high probabilities so that other failures are considered first.

3.6.5 Drift Assumptions

Not all failures result in a nullified component or a violated closed-world assumption. Consider for example a partial blockage of a pipe connecting two containers, as shown in Figure 3.16. The flow rate of liquid through the pipe is observed to be slower than expected, due to the decreased conductance of the pipe.

The set of active processes and the resulting set of influences is unaffected by such a failure, so no closed-world assumption is violated. Nullifying the path or any other component does not yield consistent predictions. Still, the system's behavior deviates from its nominal behavior

[10]This is assuming all proposition closings are treated as volatile—see Section 3.5.3.

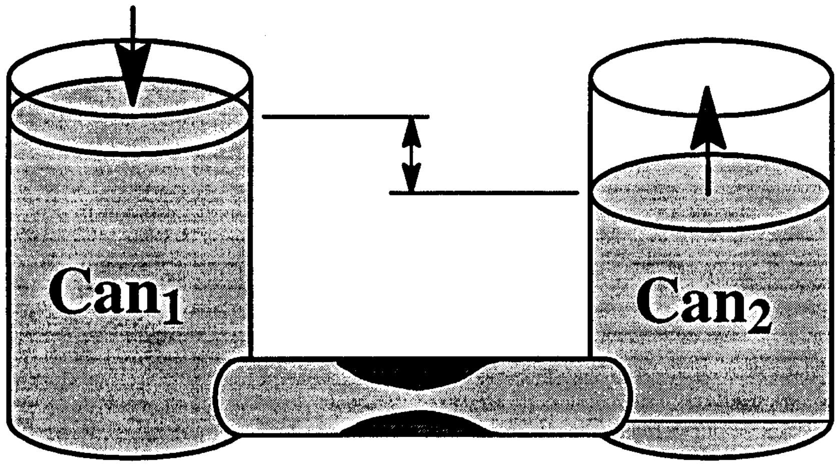


Figure 3.16: A Partially Clogged Pipe

and from our expectations. This type of failure is called a *drift failure*, indicating that some system parameter has drifted away from its nominal value.

Drift failures are generally not detectable using a purely qualitative model as a source of prediction. However, this does not imply that a qualitative model is inadequate for diagnosing the failure, once it has been detected. Drift failures do have a qualitative effect on the behavior of the system—after all, "faster", "slower", "higher" and "lower" are qualitative terms. The question is, given a qualitative model for the system, how can such qualitative perturbations in the system's outputs be traced back through the model to the system's inputs?

Reasoning about drift failures requires having a language for describing drift. That is, the qualitative modeling language must be augmented to include statements expressing drift failures. Also, the drift of related quantities must be properly constrained, so that the hypothesized drift of input parameters can propagate to explain the observed drift of system outputs.

PDE includes special assumptions, called *drift assumptions*, for each "constant" or input quantity for the system. Each such quantity is assigned two drift assumptions—one indicating a higher value than expected (positive drift), the other indicating a lower value than expected (negative drift). Naturally the default is no drift, represented as both drift assumptions being false.

In the example of the partial clog above, there are drift assumptions for the pipe's conductance (among others). An observation that the destination container is rising more slowly than expected may be explained by embracing the "negative" drift assumption for the pipe's conductance, indicating its value is lower than expected. Section 6.6 explains this example in detail.

Internally the drift of a quantity Q is represented as the sign of a special quantity ΔQ. Intuitively, ΔQ represents the difference between the actual and expected values for Q:

Definition 3 (Delta Quantity) *Let Exp(Q) be the expected value for quantity Q. The* delta *of Q, represented as $\triangle Q$, is defined as: $\triangle Q \equiv Q - Exp(Q)$.*

Thus a positive $\triangle$Q indicates that quantity Q is actually higher than originally believed; similarly, an increasing $\triangle$Q indicates that Q is increasing faster than predicted.

Each algebraic relation expressible in IQE has a corresponding constraint for the $\triangle$'s of the quantities involved. These are represented as antecedent rules, which are shown in Figure 3.17. The rules in Figure 3.17 will hold as long as a drift failure does not result in a change in the set of underlying algebraic relations. Each is derivable given that the relation holds for both actual and expected values of the quantities involved, together with the nature of $\triangle$'s as given in Definition 3. The rules for products and ratios depend on the additional assumptionthat, in the rare event where both input quantities have drifted, neither has changed sign as a result of the failure. Early detection of drift failures will ensure that these assumptions are not violated.

3.6.6 Observation Assumptions

It is always possible to explain a discrepancy between predictions and observations by challenging the observations themselves. Instruments responsible for sensing and reporting observations are prone to failure. Instruments may be included in the model so that component assumptions are included for them as well.

In general, however, observations may be wrong even though the system's instrumentation is performing flawlessly. Human observers are not flawless; sometimes our eyes or our memories deceive us. To handle such cases, explicit *observation assumptions* are introduced. For each observation, belief is conditioned on a corresponding observation assumption. Conflicts involving an observation necessarily include its assumption, as a means of questioning and retracting belief in "what we think we saw". Typically observation assumptions are assigned high probability relative to other defeasible assumptions (see Section 4.3.4.1), so that observations are brought into question only when nothing else makes sense.

3.7 Discussion

This chapter has presented a language for expressing qualitative models for diagnosing continuous systems, based on the ideas of QP theory. Domain theories and domain-independent constraints are both represented in terms of rules rather than opaque procedures. These rules may be run forward during simulation and prediction, or searched backward for explanations of unexpected behaviors. An inspectable representation of domain-independent constraints is essential to having a uniform framework for searching for abductive explanations.

The unique compositional nature of the process-centered models of QP theory requires the explicit representation of closed-world assumptions, which may be violated by a failure. By including closed-world assumptions in the set of culpable beliefs and allowing the candidate generator to retract them as needed, we are able to postulate structure missing from the scenario description—something which traditional consistency-based approaches cannot do.

Other types of defeasible assumptions provide additional tools which may be used to repair a broken model. Retraction of modeling assumptions permits a change of perspective or a refinement to an overly-simplified model. Mode assumptions capture internal state and operating regions of components, and may be used to represent common modes of failure. Domain

```
;;; Translations to Deltas:
(=TR=>  (High ?Q)  (Pos (Delta ?Q)))
(=TR=>  (Low  ?Q)  (Neg (Delta ?Q)))
(=TR=>  (Q_Ok ?Q)  (Zip (Delta ?Q)))
(=TR=>  (Fast ?Q)  (Inc (Delta ?Q)))
(=TR=>  (Slow ?Q)  (Dec (Delta ?Q)))
(=TR=>  (D_Ok ?Q)  (Con (Delta ?Q)))

(=TR=>  (Slowly_Increasing ?Q)  (and (Inc ?Q) (Dec (Delta ?Q))))
(=TR=>  (Slowly_Decreasing ?Q)  (and (Dec ?Q) (Inc (Delta ?Q))))
(=TR=> (Rapidly_Increasing ?Q)  (and (Inc ?Q) (Inc (Delta ?Q))))
(=TR=> (Rapidly_Decreasing ?Q)  (and (Dec ?Q) (Dec (Delta ?Q))))

;;; Rules constraining Deltas
(==> (M+ ?Q1 ?Q2)  (M0+ (Delta ?Q1) (Delta ?Q2)))
(==> (M- ?Q1 ?Q2)  (M0- (Delta ?Q1) (Delta ?Q2)))

(==> (Deriv  ?Q1 ?Q2)  (Deriv  (Delta ?Q1) (Delta ?Q2)))
(==> (Qprop+ ?Q1 ?Q2)  (Qprop+ (Delta ?Q1) (Delta ?Q2)))
(==> (Qprop- ?Q1 ?Q2)  (Qprop- (Delta ?Q1) (Delta ?Q2)))

(==> (Sum_Member ?Q1 ?Q2 ?s)  (Sum_Member (Delta ?Q1) (Delta ?Q2) ?s))

(==> (and (:= ?Q1 ?Q2) (:TEST (not (numeric-op? ?Q2))))
     (:=  (Delta ?Q1) (Delta ?Q2)))

(==> (:= ?Q1 (+ ?Q2 ?Q3))  (:= (Delta ?Q1) (+ (Delta ?Q2) (Delta ?Q3))))
(==> (:= ?Q1 (- ?Q2 ?Q3))  (:= (Delta ?Q1) (- (Delta ?Q2) (Delta ?Q3))))

(==> (:= ?prod (* ?M1 ?M2))
     (and (==> (and (Pos ?m1) (Pos (Delta ?m2)))  (Affected (Delta ?prod) +))
   (==> (and (Pos ?m2) (Pos (Delta ?m1)))  (Affected (Delta ?prod) +))
   (==> (and (Neg ?m1) (Neg (Delta ?m2)))  (Affected (Delta ?prod) +))
   (==> (and (Neg ?m2) (Neg (Delta ?m1)))  (Affected (Delta ?prod) +))
   (==> (and (Pos ?m1) (Neg (Delta ?m2)))  (Affected (Delta ?prod) -))
   (==> (and (Pos ?m2) (Neg (Delta ?m1)))  (Affected (Delta ?prod) -))
   (==> (and (Neg ?m1) (Pos (Delta ?m2)))  (Affected (Delta ?prod) -))
   (==> (and (Neg ?m2) (Pos (Delta ?m1)))  (Affected (Delta ?prod) -)))

(==> (= ?rat (/ ?N ?D))
     (and (==> (and (Pos ?N) (Pos (Delta ?D)))  (Affected (Delta ?rat) -))
   (==> (and (Pos ?D) (Pos (Delta ?N)))  (Affected (Delta ?rat) +))
   (==> (and (Neg ?N) (Neg (Delta ?D)))  (Affected (Delta ?rat) -))
   (==> (and (Neg ?D) (Neg (Delta ?N)))  (Affected (Delta ?rat) +))
   (==> (and (Pos ?N) (Neg (Delta ?D)))  (Affected (Delta ?rat) +))
   (==> (and (Pos ?D) (Neg (Delta ?N)))  (Affected (Delta ?rat) -))
   (==> (and (Neg ?N) (Pos (Delta ?D)))  (Affected (Delta ?rat) +))
   (==> (and (Neg ?D) (Pos (Delta ?N)))  (Affected (Delta ?rat) -)))
```

Figure 3.17: Rules for Propagating Drift

theory assumptions extend the diagnostic engine into the realm of theory revision. Drift assumptions propagate the effects of an increase or decrease in an input parameter, despite a lack of a qualitative change in behavior. Observation assumptions provide a way out in those rare cases where the observations themselves are in error. These assumptions combine to provide significant flexibility and robustness in diagnosing and repairing a failed model.

The next chapter describes how the predictions of the model are used to guide the search for plausible diagnostic candidates. Chapter 5 discusses the use of abductive search as a means of explaining closed-world assumption violations.

Chapter 4

Candidate Generation

4.1 Introduction

One of the primary areas of focus for this research—and for model-based diagnosis research in general—has been the generation of candidate diagnoses. Candidate generation refers to the process of exploring the space of diagnostic hypotheses in search of an explanation for some observed symptomatic behavior.

Davis and Hamscher [14] decompose model-based diagnosis into three phases: candidate [hypothesis] generation, test, and discrimination. The test phase is intended to weed out candidates which do not agree with the symptoms. The discrimination phase chooses among the surviving candidates, based on some measure of the candidates' credibility, possibly after additional observations have been made.

A trivial candidate generator simply enumerates the entire space of hypotheses in some arbitrary order. As long as the generator is complete, it will eventually find the correct candidate. Unfortunately, the space of hypotheses grows exponentially with the complexity of the system; thus this simple generator is too inefficient for most serious diagnostic problems.

To be efficient, a candidate generator must avoid exploring the subspace of hypotheses which do not explain the symptoms. In the language of the AI paradigm of generate and test, the test must be incorporated into the generator. In addition, consideration of the remaining subspace must be reasonably ordered so as to maximize the chances of finding the correct candidate early in the search. These two issues—avoiding inconsistency and best-first ordering of search—are the primary dimensions along which candidate generation research varies.

This chapter describes several contributions to candidate generation. Section 4.2 provides an overview of candidate generation as formalized and implemented by other researchers in model-based diagnosis. Section 4.3 defines a modified version of Reiter's *DIAGNOSE* algorithm for computing diagnoses as hitting sets. Section 4.4 describes an efficient Assumption-based Truth Maintenance System, and shows how it is particularly well-suited for candidate generation. Finally, Section 4.5 summarizes the contributions of this research in the area of candidate generation.

4.2 Candidates and Conflicts

This section outlines the most relevant prior research in model-based candidate generation. Rather than echo descriptions and definitions verbatim, the terminology introduced by other researchers is merged into a consistent compilation. This provides a unifying view of the work upon which this research builds and facilitates the discussion of the contributions of this research.

The goal of diagnosis is to suggest possible *explanations* for a discrepancy between observed and predicted behavior. An explanation consists of a description of the failed system, indicating (at least) which parts of the system are behaving according to the model and which are not. It is the nature of the diagnostic task that multiple explanations may be consistent with a given set of symptoms—at most one of which is correct. A potential explanation has been variously referred to as a *(diagnostic) hypothesis*, or *(candidate) diagnosis*, or simply *candidate*. In this discussion the term "diagnosis" is reserved for the overall endeavor, and the term "candidate" is used to refer to a particular hypothesis.

A candidate in traditional model-based diagnosis refers generally to a set of *status assignments*, one for each component in the system. Only those candidates which are consistent with observed symptomatic behavior provide viable explanations. Unless stated otherwise, the term "candidate" may be assumed to imply consistency.

Additional observations, or *probes* may be required to distinguish among competing candidates. A number of researchers have investigated strategies for probe selection [26, 14], utilizing results from information and decision theory. The expected benefits of a probe sequence are compared against the expected costs and risks. A related area of research involves *test generation* [14]: the selection of new input values such that observable outputs will best distinguish among the candidates. Probe selection and test generation are important areas of ongoing investigation; however, they are not the focus of this research. Still, nearly all of the advances made in these areas apply equally well to the diagnostic approach discussed here.

4.2.1 Consistency-Based Diagnosis

In its simplest form—and in most of the early research in model-based diagnosis—the status of a component is represented by a boolean value indicating either that it is working properly or not. Most early models only specified constraints for working components; a failed component could exhibit arbitrary behavior. This approach to candidate generation is generally referred to as "consistency-based diagnosis", because it finds models which are consistent with rather than predictive of observed symptoms. Prominent within this approach is a technique developed by Davis [13] called "constraint suspension", whereby individual constraints associated with suspected components are temporarily suspended in an attempt to remove the conflict between predicted and observed behavior.

Within the constraint suspension approach, and in fact most approaches in consistency-based diagnosis, every component is either working correctly or not. This allows a candidate to be compactly represented by the set of suspected components. Under this representation, every superset of a candidate is also a candidate, because suspending additional constraints cannot introduce an inconsistency. This monotonicity allows the set of all candidates to be represented parsimoniously by the set of minimal[1] candidates. For this reason, and because minimal candi-

[1]Minimal is defined w.r.t. set inclusion.

dates provide more plausible explanations than their supersets, candidate generation searches only for minimal candidates.[2] This greatly reduces the number of candidates to be considered.

Still, there will often be too many minimal candidates to consider them all. Some researchers [13, 46] have avoided this problem by only considering single failures. A more general solution finds and returns candidates best-first, where "best" is determined by size or some other estimate of likelihood. Candidates are generated and evaluated until some stopping criterion is met. This is the approach taken in this research.

4.2.2 Candidates as Hitting Sets

Reiter [68] provides a formalization of model-based diagnosis which has been adopted by much of the diagnosis research community. Following the constraint suspension framework, Reiter considers a candidate to be a set of component failures which could account for all discrepancies between observations and expectations. The failure of a component c is represented by the predicate $AB(c)$, which indicates that c is in some way *abnormal*.[3] The correctness of component c is indicated by $\neg AB(c)$. The forms $AB(c)$ and $\neg AB(c)$ are called *AB-literals*; $AB(c)$ is a positive AB-literal and $\neg AB(c)$ is a negative AB-literal. The set of all negative AB-literals represents the set of default beliefs, and is designated $\mathcal{D}$.

A candidate represents a set of default beliefs to be retracted. Retracting belief that the candidate components are working correctly must eliminate all inconsistent predictions of the model. A conjunction of beliefs which leads to inconsistent predictions is called a *conflict set*, or *conflict*:

Definition 4 (Conflict Set) *Let S_c be a conjunctive set of AB-literals. S_c is a* conflict set *for theory $\mathcal{T}$ and observations OBS iff $S_c \cup \mathcal{T} \cup OBS$ is unsatisfiable. A minimal conflict set is a conflict set which contains no smaller conflict set.*

For purposes of the current discussion,[4] a conflict is a set of components which cannot all be working correctly, given the model and observations. In order to be consistent, a candidate must "hit" every conflict—that is, it must suspect at least one component from each conflict. The concept of a *hitting set* is central to candidate generation:

Definition 5 (Hitting Set) *Suppose $\mathcal{S}$ is a set of sets. A set S_h is a* hitting set *for $\mathcal{S}$ iff $\forall S_i \in \mathcal{S} : S_h \cap S_i \neq \emptyset$. A minimal hitting set is a hitting set for which no proper subset is a hitting set.*

In particular, we are interested in the hitting sets of the set of conflicts—referred to here as the *conflict-hitting sets*:

Definition 6 (Conflict-Hitting Set) *Let $\mathcal{S}_C$ be the set of all conflicts (Definition 4). A set S_h is a* conflict-hitting set *iff S_h is a hitting set for $\mathcal{S}_C$.*

[2]Reiter [68] includes minimality in his conditions for candidacy; for him, "non-minimal candidate" is an oxymoron. Allowing for non-minimal candidates here simplifies later discussion.

[3]This is the same abnormality predicate commonly used in circumscription and nonmonotonic logic.

[4]The more general definition is given here for compatibility with the types of defeasible assumptions introduced in Section 3.6.

Within traditional consistency-based diagnosis, both conflicts and conflict-hitting sets consist of correctness assumptions for components (i.e., the negative AB-literals), sometimes represented simply as sets of components. If we view a candidate as a set of retracted beliefs about the correctness of certain components, then every candidate must be a conflict-hitting set. Otherwise, the components of some conflict set are believed to be working correctly, leading to inconsistent predictions. Also, any set of components which hits every conflict must be a consistent candidate, because the remaining components would not be a superset of any conflict. Thus, being a conflict-hitting set is both a necessary and sufficient condition for candidate consistency:

Theorem 2 (Candidate) *A set $S \subseteq \mathcal{D}$ is a candidate diagnosis iff S is a conflict-hitting set.*

Because every conflict is a superset of some minimal conflict, any set that covers the minimal conflicts is a conflict-hitting set. Thus only minimal conflicts need to be considered when generating or checking candidates. Section 4.3 describes Reiter's *DIAGNOSE* algorithm for generating minimal hitting sets, along with some significant refinements.

4.2.3 The General Diagnostic Engine (GDE)

De Kleer and Williams [25, 26] cast candidate generation in terms of ATMS constructs. Their General Diagnostic Engine, or GDE, uses the ATMS for predicting consequences and detecting inconsistencies of diagnostic candidates. Each component has an associated ATMS assumption indicating that the component is working according to the model. The assumption is used to justify the constraints of the component. ATMS label propagation provides the minimal sets of working components required for each prediction sanctioned by the model.

Whenever a prediction conflicts with an observation or another prediction, an ATMS *nogood* results. A nogood is a minimal set of assumptions which cannot all be true—a conflict set using Reiter's terminology. A nogood may be represented either as a negated conjunction of assumptions (a contradictory environment), or as a disjunction of negated assumptions—what de Kleer calls a *choose*. Given this latter view, it is clear that resolving a nogood requires satisfying the corresponding choose, by selecting one (or more) of its negated assumptions.

The set of minimal nogoods is computed and cached automatically during normal ATMS label propagation. These minimal nogoods are utilized by GDE during candidate generation. Again, a candidate is a set of component assumptions which covers the nogoods—that is, whose retraction eliminates all nogoods. Finding all minimal candidates requires computing the *Cartesian product* of the nogood chooses. In effect the conjunctive normal form (cnf) of the minimal nogood chooses is converted to disjunctive normal form (dnf) and then simplified. This is the same computation that the ATMS performs when constructing interpretations and when propagating labels through a multiple-antecedent justification.

Conceptually the ATMS provides a very natural framework for candidate generation. Unfortunately, the GDE algorithm outlined above suffers from a major inefficiency: it requires explicitly enumerating the set of minimal nogoods, which may be exponentially large. This requirement was later eliminated in a descendant of GDE, called *Sherlock*.

4.2.4 Diagnosis with Behavioral Modes: *Sherlock*

Two limitations of prior work in model-based diagnosis have become clear in recent years. First, many types of components behave differently depending on their current mode of operation. For example, a properly-functioning transistor may be in one of the following modes: {*off,linear,saturated*}. Failure of an upstream component may cause a transistor to operate in a different mode than predicted by the model, even though there is nothing wrong with the transistor itself. Second, many types of components tend to fail in only a few ways, some of which are more likely than others. For example, a failed relay is most likely to be in a stuck-open condition, but could be stuck-closed.

The approach to candidate generation described thus far ignores these important issues. De Kleer and Williams [27] extended GDE to accommodate *behavioral modes*, resulting in a new system called *Sherlock*. For each combination of component and mode, Sherlock builds a *mode assumption* which justifies the constraints defining that mode of behavior. Sherlock performs a best-first search of the space of candidates, where a candidate consists of mode assignments for the set of components. Sherlock avoids the up-front cost of full prediction and conflict detection suffered by GDE, by focusing on one candidate at a time. A similar focusing strategy is used in PDE and is discussed in Section 4.4.

In specifying the set of behavioral modes for a component, there is an implicit *closed-world assumption* that the modes represented are in fact the only possible modes for that component. This closed-world assumption may be explicitly represented as a defeasible assumption. This is in effect what Sherlock does. By including an additional *unknown mode* in the set of possible modes for each component, Sherlock does not sacrifice the ability to reason with unanticipated modes of failure. The unknown mode has no model; that is, there are no consequences of the mode assumption for the unknown mode. Instead it provides the ability to suspend all constraints for the component, as in the standard consistency-based approach.

Including an unknown mode in the set of possible modes for a component only makes sense when probabilities or some other ordering metric are available. This is because the unknown mode is always a consistent explanation for any failure (or even non-failure). Because the unknown mode has no logical consequences, it will never lead to prediction or corroboration of the symptoms, thus making it less credible as a diagnosis. The unknown mode is typically assigned a low probability relative to the more specific modes, so as to prefer more specific explanations.

4.3 Computing Hitting Sets

This section describes an algorithm for generating the minimal hitting sets for the set of conflicts. This algorithm is based on Reiter's *DIAGNOSE* algorithm [68], but with three important improvements. First, it avoids much of the redundant search and resulting inefficiency of Reiter's algorithm. Second, it is not limited to the constraint suspension view of candidate generation, but instead generalizes to closed-world assumptions and other defeasible assumptions, as introduced in Section 3.6. Third, this algorithm deals with the reality of incompleteness in the theorem prover, providing a balance between the search costs in the two systems.

This section begins with an overview of Reiter's *DIAGNOSE* algorithm for computing hitting sets, and then describes a variation which avoids redundant search. The domain of the algorithm is then extended beyond the constraint suspension approach, by applying it to a more general view of candidate generation which includes behavioral modes. Finally the algorithm is modified again to minimize the effects of incompleteness in the theorem prover.

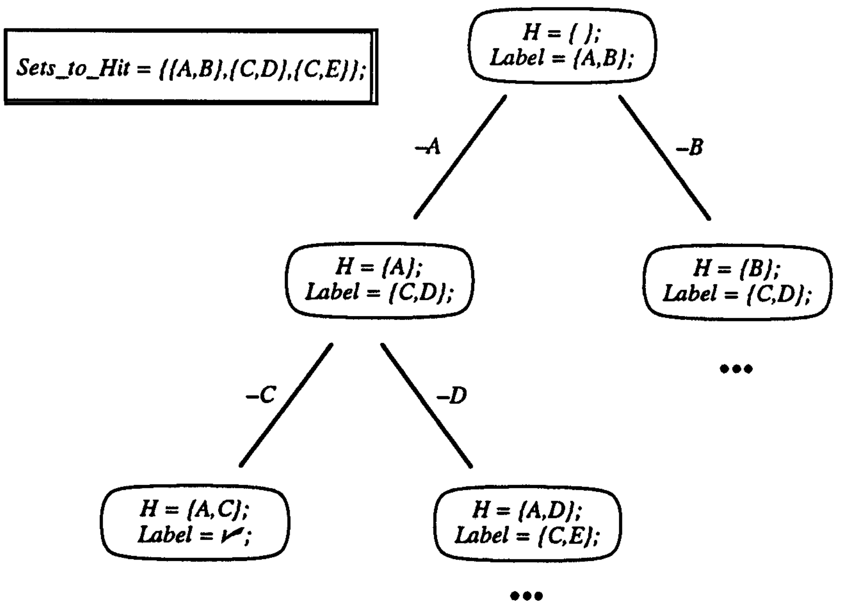


Figure 4.1: The *$\mathcal{HS}$-Tree* Data Structure

4.3.1 Reiter's *DIAGNOSE* Algorithm

In [68], Reiter defines an algorithm for finding minimal hitting sets. The *DIAGNOSE* algorithm has several desirable properties:

1. It generates all minimal hitting sets;
2. It generates smallest hitting sets first;
3. It does not require enumerating all minimal conflict sets;
4. It prunes (some) redundant search paths.

4.3.1.1 The *$\mathcal{HS}$-tree* Data Structure

Reiter defines the *DIAGNOSE* algorithm around a data structure which he calls an *$\mathcal{HS}$-tree*, as depicted in Figure 4.1. Given a collection of sets $\mathcal{S}_{\mathcal{H}}$, an *$\mathcal{HS}$-tree* for $\mathcal{S}_{\mathcal{H}}$ is a node-labeled and edge-labeled tree satisfying the following properties. For any node n in the tree, $H(n)$ denotes the collection of edge labels on the path to n from the root node. Each node n has a label *Label*(n), which is a set Σ in $\mathcal{S}_{\mathcal{H}}$ such that $\Sigma \cap H(n) = \emptyset$, if such a set exists; otherwise *Label*$(n) = \surd$ and n is a leaf node. If n is labeled by Σ in $\mathcal{S}_{\mathcal{H}}$, then for each σ in Σ, n has a successor node n_σ joined to n by an edge labeled σ.

Reiter proves that for every node n labeled $\surd$ in an *$\mathcal{HS}$-tree* for $\mathcal{S}_{\mathcal{H}}$, $H(n)$ is a hitting set for $\mathcal{S}_{\mathcal{H}}$; and for every minimal hitting set S, there is some node n labeled $\surd$ such that $H(n) = S$. In other words, the leaves of the *$\mathcal{HS}$-tree* include all minimal hitting sets, and possibly some non-minimal hitting sets as well.

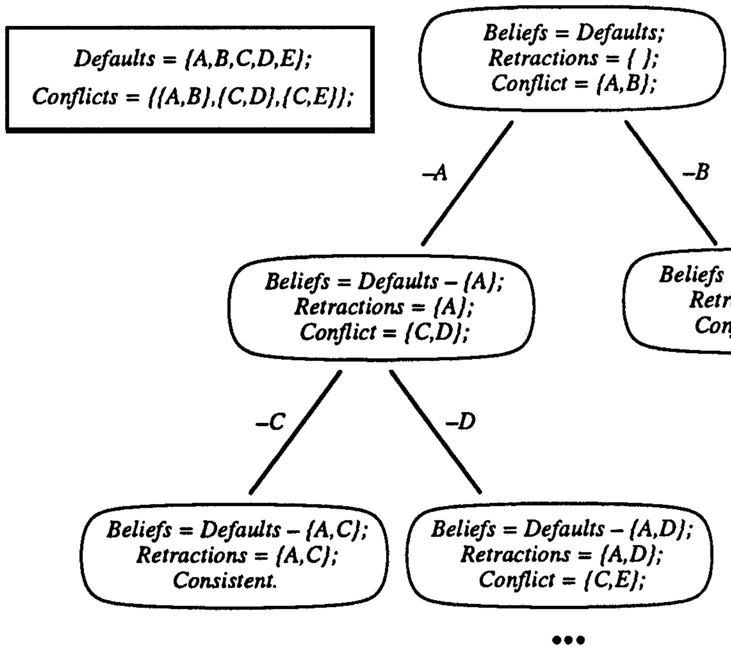


Figure 4.2: A Default Reasoning Interpretation of the *$\mathcal{HS}$-Tree*

4.3.1.2 Default Reasoning and Conflict-Hitting Sets

The *$\mathcal{HS}$-tree* structure is very general in that it finds the minimal hitting sets for any arbitrary collection of sets. For candidate generation, we are interested in finding minimal hitting sets of the set of conflicts. In traditional consistency-based diagnosis, conflicts consist of sets of components; and (minimal) conflict-hitting sets represent (minimal) retractions of belief about the correctness of components, such that the remaining components yield consistent predictions. The correctness of a component is a *default belief*, and the goal of candidate generation is to retract as few defaults as possible to restore consistency.

Given these specifics, the *$\mathcal{HS}$-tree* in Figure 4.1 may be interpreted as shown in Figure 4.2. Each node in the *$\mathcal{HS}$-tree* may be viewed as representing a set of default beliefs not yet retracted:

Definition 7 (Beliefs, Retractions and Conflicts) *Suppose n is a node in the $\mathcal{HS}$*-tree *for the set of conflicts $\mathcal{S}_C$ contained in the set of default beliefs $\mathcal{D}$. The (partial) hitting set denoted by $H(n)$ designates a set of* retractions *from $\mathcal{D}$; in this context $H(n)$ is renamed Retracts(n). The* belief set *for n, denoted by Beliefs(n), represents the remaining beliefs in $\mathcal{D}$ after performing retractions: Beliefs$(n) \equiv \mathcal{D}$ – Retracts(n). The label of n, previously denoted by Label(n), is renamed Conflict(n), and represents a conflict in $\mathcal{S}_C$ which is a subset of Beliefs(n), if such a conflict exists.*

The root node of the *$\mathcal{HS}$-tree* represents the set of all default beliefs. For any node n, if *Beliefs*(n) is consistent, then *Conflict*$(n) = \surd$ (indicating no conflicts were found), n is a leaf node and *Retracts*(n) is a conflict-hitting set. Otherwise, *Conflict*$(n) = \Sigma$, where Σ is some minimal conflict contained in *Beliefs*(n). Every conflict-hitting set must intersect with every

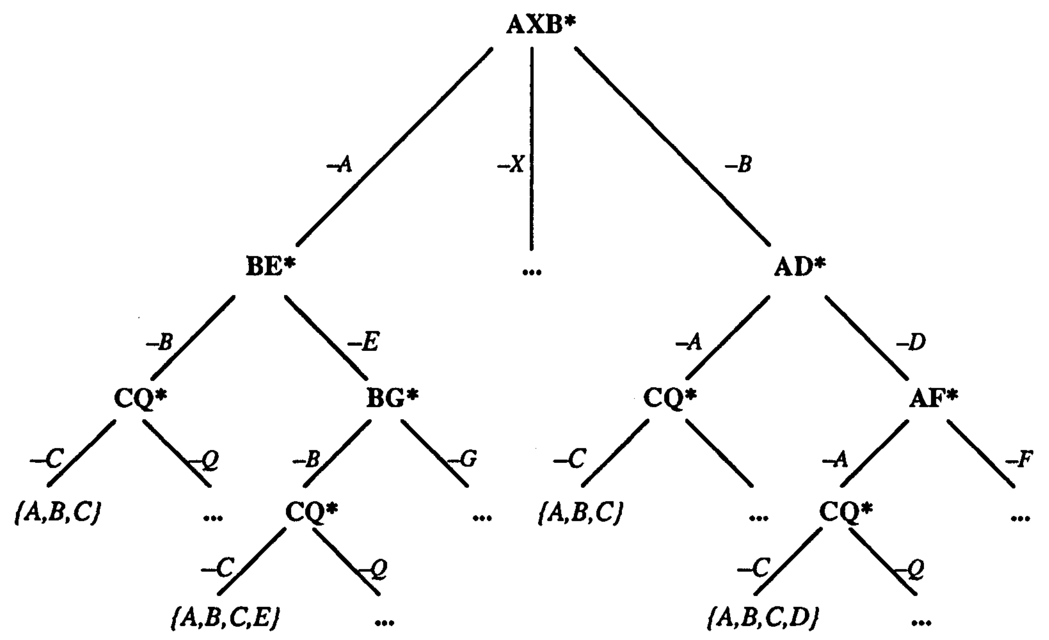


Figure 4.3: Redundancy in an $\mathcal{HS}$-*Tree*

conflict including Σ, so at least one default belief in Σ must be retracted. The $\mathcal{HS}$-*tree* contains a separate subnode below n for each of the ways of hitting Σ.

The basic algorithm for generating the $\mathcal{HS}$-*tree* falls out naturally given this perspective. The root node is created with the global set of default beliefs and no retractions: *Beliefs*(*root*) $\equiv \mathcal{D}$ and *Retracts*(*root*) $\equiv \emptyset$. At each node n in the tree, *DIAGNOSE* looks for some minimal conflict Σ contained in *Beliefs*(n). If such a conflict is found, then *Beliefs*(n) is still inconsistent and some default in Σ must be retracted. For each literal σ in Σ, the tree is extended by a node n_σ representing the retraction of σ from *Beliefs*(n):
Beliefs(n_σ) $\equiv$ *Beliefs*(n) $- \{\sigma\}$, and *Retracts*(n_σ) $\equiv$ *Retracts*(n)$\cup\{\sigma\}$.

In effect the algorithm is using the conflicts it finds as a basis for retracting beliefs. When enough retractions have occurred to remove the inconsistency (i.e., no conflicts are contained in *Beliefs*(n)), then the cumulative set of retracted beliefs *Retracts*(n) represents a conflict-hitting set.

4.3.1.3 Pruning Redundant Paths

The algorithm as defined so far is guaranteed to find all minimal hitting sets for the conflicts contained in the set of defaults $\mathcal{D}$. By growing the tree breadth-first, the algorithm will never find a proper subset of a hitting set it has already found. However, it may find the same hitting set more than once, and finds non-minimal hitting sets—that is, supersets of known hitting sets.

Figure 4.3 illustrates the potential redundancy in the $\mathcal{HS}$-*tree*. Note that hitting set $\{A, B, C\}$ is found twice, and supersets $\{A, B, C, D\}$ and $\{A, B, C, E\}$ are found as well. This redundant search adds dramatically to the cost of finding hitting sets. Fortunately, much of the redundancy is easily avoided.

In order to reduce redundant search, Reiter introduces two pruning heuristics.[5] The first heuristic prunes any branch whose cumulative retractions are a superset of a known hitting set. Such a branch will necessarily yield a single non-minimal hitting set; checking for subsumption early on avoids the cost of a consistency check, which is considered to be high. This heuristic prunes non-minimal hitting sets as they are discovered, but does not eliminate all redundant branches. The second heuristic prunes any branch whose partial hitting set (i.e., cumulative set of retractions) is identical to that of some other branch in the tree. This is justified because the two subtrees are identical in terms of their potential for generating hitting sets. While this pruning heuristic avoids some of the redundancy suffered by the basic algorithm for constructing an $\mathcal{HS}$-*tree*, it is still possible to find (and prune) the same hitting set multiple times.

4.3.2 Protection Sets

After careful examination of the algorithm and the nature of the problem, it became apparent that the inefficiency described above was unnecessary. Here we describe an approach for pruning redundant branches which subsumes Reiter's second heuristic, while pruning significantly more of the tree. In addition, this improved algorithm avoids the need for maintaining a data structure (such as a hash table or discrimination tree) for indexing equivalent partial hitting sets.

Rather than relying on Reiter's second pruning heuristic, this algorithm delegates responsibility for finding different hitting sets among the children of each node in the $\mathcal{HS}$-*tree*. Each child node and its descendants are prevented from making retractions which have been assigned to other branches of the $\mathcal{HS}$-*tree*. These protected beliefs are maintained for each node in the tree, and are referred to as a node's *protection set*:

Definition 8 (Protection Set) *Suppose n is a node in an $\mathcal{HS}$*-tree. *The* protection set *for n, designated Protects(n), is defined as follows: If n is the root of the tree, then Protects(n) $\equiv$ $\emptyset$; otherwise let p be the parent node of n, and σ_i be the label of the edge from p to n:*
Protects(n) $\equiv$ Protects(p) $\cup$ $\{\sigma_h \mid \sigma_h \in$ Conflict(p), $h < i\}$.

The hitting set algorithm is modified to compute and utilize protection sets as follows. At each node n in the $\mathcal{HS}$-*tree*, the algorithm performs an "upper triangulation" of the branches forming the subtree at n. The elements of *Conflict*(n) are ordered arbitrarily, and the first is used to spawn a subnode, as described above. This first branch simply inherits the protection set from the parent node. The protection set for each subsequent branch is the union of the parent's protection set and the labels of the alternative branches taken thus far.

A protected belief for a given branch may not be retracted, and so will not participate in any hitting set found within that branch. The basic idea is that some other branch of the search is responsible for finding hitting sets containing a protected belief. Thus any intersection between a node's conflict set and its protection set represents a pruned branch; if the conflict set is completely contained in the protection set, then all branches from that node are pruned. A branch which may be pruned due to collision with a protection set is called a *protection violation*:

[5] Reiter included a third heuristic for dealing with non-minimal conflicts; we ignore this possibility here, since it is easy to find some minimal conflict within a non-minimal one; see Section 4.4.3.1.

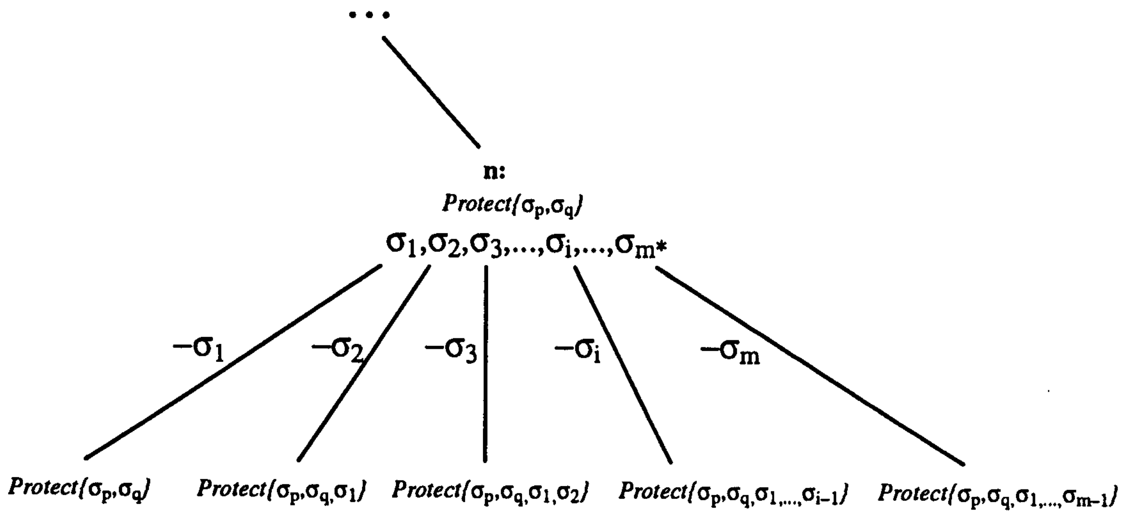


Figure 4.4: Using Protection Sets in the Search for Hitting Sets

Definition 9 (Protection Violation) *A node n in an $\mathcal{HS}$-*tree *constitutes a* protection violation *exactly when Protects(n)$\cap$Retracts(n) $\neq \emptyset$.*

Proposition 1 (Monotonicity of Protection Violations) *Let n be a node in an $\mathcal{HS}$-*tree. *If n is a protection violation, then every node in the subtree rooted at n is a protection violation.*

Proof: Both protection sets and retraction sets grow monotonically from parent node to child node. If these two sets intersect for the parent node, they must intersect for every child node as well. □

Proposition 2 (Protected Beliefs) *Suppose n is a node in an $\mathcal{HS}$-*tree. *If n is not a protection violation, then Protects(n) $\subset$ Beliefs(n).*

Proof: By Definition 9, *Protects*$(n)\cap$*Retracts*$(n) = \emptyset$. Because *Protects*$(n) \subset \mathcal{D}$, it follows that *Protects*$(n) \subset [\mathcal{D} -$ *Retracts*$(n)]$. By Definition 7, *Beliefs*$(n) \equiv \mathcal{D} -$ *Retracts*(n), so *Protects*$(n) \subset$ *Beliefs*(n). □

Consider the partial $\mathcal{HS}$-*tree* shown in Figure 4.4. Node n is in the process of being extended to complete the subtree rooted at n. Node n is labeled by a size-m conflict set, *Conflict*$(n) = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$, and so n has m branches below it. The first branch (labeled σ_1) is responsible for finding all hitting sets containing *Retracts*$(n)\cup\{\sigma_1\}$. The second branch finds all hitting sets containing *Retracts*$(n)\cup\{\sigma_2\}$ but not containing σ_1. In general, the *ith* branch finds hitting sets containing *Retracts*$(n)\cup\{\sigma_i\}$ but none of $\{\sigma_1, \sigma_2, \ldots, \sigma_{i-1}\}$.

Theorem 3 (Protected $\mathcal{HS}$-tree) *Suppose T is an $\mathcal{HS}$-*tree *for $S_{\mathcal{H}}$. Let T' be the tree resulting after pruning all non-minimal hitting sets and protection violations in T. T' is called a* protected $\mathcal{HS}$-*tree* for $S_{\mathcal{H}}$. *The set of leaf nodes labeled by $\surd$ in T' correspond one-to-one to the set of minimal hitting sets for $S_{\mathcal{H}}$.*

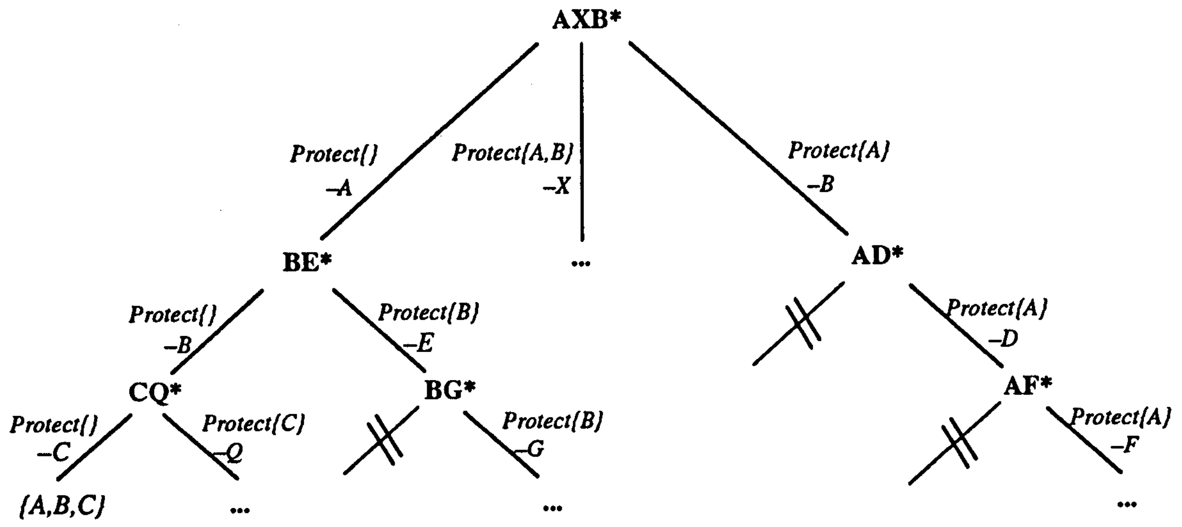


Figure 4.5: The Result of Adding Protection Sets to the *HS-Tree* in Figure 4.3

Proof: Because T' is a subset of the original $\mathcal{HS}$-*tree* T, it follows that all of its leaf nodes labeled by $\surd$ represent minimal hitting sets (see [68] for proof). By Proposition 1, protection violations are passed on from parent node to child node. Thus pruning all protection violations has the same effect as pruning only leaf node protection violations. It remains to prove that a leaf node is a protection violation if and only if it constitutes a redundant hitting set.

$\Longleftarrow$: First we show that a protected $\mathcal{HS}$-*tree* contains no duplicate hitting sets. We show that every node in the tree guarantees that each of its subtrees finds a disjoint set of hitting sets. By protecting a retraction from each of the previous $i-1$ branches, the *ith* branch is guaranteed not to find duplicate hitting sets found by the previous branches. Similarly, no later branch will find a hitting set found by the *ith* branch, because they will all protect σ_i from being retracted. Because any two leaves must have a nearest common ancestor in the tree, there is no way for two leaves to represent the same hitting set. Thus every hitting set found must be unique.

$\Longrightarrow$: Now we show that only redundant hitting sets are pruned. Consider a leaf node n_l in T which is pruned from T'. Let S_H be the minimal hitting set represented by *Retracts*(n_l). By Definition 9, *Protects*$(n_l) \cap S_H \neq \emptyset$. Let n_c be the highest node (i.e., closest to the root) on the path from the root to n_l such that *Protects*$(n_c) \cap S_H \neq \emptyset$. Let n_P be the parent of n_c. By assumption, *Protects*$(n_p) \cap S_H = \emptyset$. Let *Conflict*$(n_p) = \{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ be the ordered branches from n_p, and let σ_i be the branch to n_c. By Definition 8, [*Protects*$(n_c)-$*Protects*(n_p)] $= \{\sigma_1, \sigma_2, \ldots, \sigma_{i-1}\}$, and this set must intersect with S_H. Let σ_j be the first such protection: $j = min_{k=1}^{i-1} k : \sigma_k \in [S_H \cap$ *Protects*$(n_c)]$. The *jth* branch does not protect σ_i or any other literals in S_H; therefore S_H will be found below the *jth* branch. Thus only redundant branches are pruned by protections. $\square$

Each protected belief P is introduced only after some other sibling node n_j has already taken care of retracting P. Thus any hitting set containing P (and the retractions already made) will be found in the subtree rooted at n_j, and nowhere else.

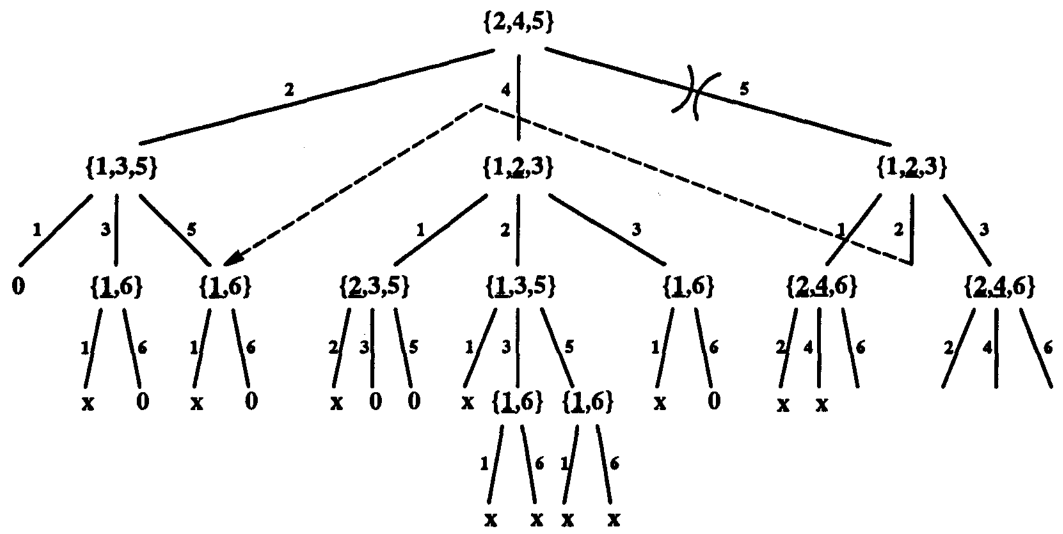


Figure 4.6: Reiter's *HS-Tree* for the conflicts: {{245}{123}{135}{246}{24}{235}{16}}.

The use of protection sets can result in a significant reduction in the size of the *HS-tree*. Figure 4.5 shows the protected *HS-tree* resulting from applying protection set pruning to the *HS-tree* in Figure 4.3. Note that all of the redundant branches are pruned. The use of protection sets as defined above subsumes Reiter's pruning heuristic which looks for equivalent partial hitting sets. It is impossible for a later branch to produce the same (partial) hitting set as produced by an earlier branch, because the later branch is prevented from making the first retraction of the earlier branch. This algorithm also prunes in many cases where Reiter's heuristic does not; still, it is necessary to prune search when partial hitting sets grow to contain a known hitting set.[6] The cost of performing this check grows linearly in the number of hitting sets found, and is thus reasonably efficient.

Figures 4.6 and 4.7 show the *HS-trees* constructed by Reiter's *DIAGNOSE* algorithm and the algorithm presented here, given the following conflicts:
$\{\{2,4,5\},\{1,2,3\},\{1,3,5\},\{2,4,6\},\{2,4\},\{2,3,5\},\{1,6\}\}$. This example was originally used by Reiter [68] to demonstrate his algorithm. While part of the complexity of Reiter's *HS-tree* (the right branch in Figure 4.6) is due to the presence of the non-minimal conflicts $\{2,4,5\}$ and $\{2,4,6\}$, several other branches are avoidable using protection sets. The protected beliefs are underlined in both figures. Even in this simple example, protection sets are able to prune better than half of the *HS-tree*.

Analysis of the complexity of using protection sets is hampered by the dependence on the distribution of conflict sizes relative to the total number of beliefs. Larger conflicts lead to greater savings, as do larger hitting sets (i.e., the depth of the tree). Due to the exponential nature of the *HS-tree*, any reduction in branching factor yields a significant decrease in the overall size of the tree. Empirical results show an order of magnitude improvement in performance for problems of moderate size.

[6] A "left" branch can find a superset of a minimal hitting set found down a "right" branch, but not vice-versa.

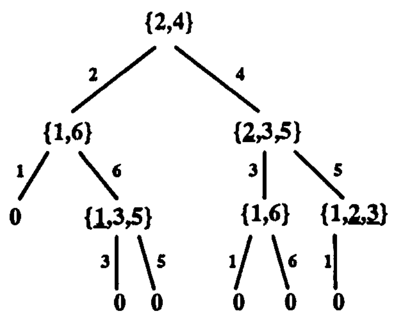


Figure 4.7: A protected *$\mathcal{HS}$-Tree* for the conflicts: {{245}{123}{135}{246}{24}{235}{16}}.

4.3.2.1 Favoring Mostly-Protected Conflicts

As noted above, the protection sets are used to avoid redundant search by preventing certain retractions down each branch of the *$\mathcal{HS}$-tree*. If a node's conflict set and protection set have a non-empty intersection, then would-be branches within this intersection need not be searched. In particular, if the entire conflict set falls within the protection set, then no further search is required down this path. This is because the set of beliefs for the node contains a minimal conflict which is protected, and so cannot be removed.

This suggests that the algorithm should look for conflicts which overlap with the protection set as much as possible. More accurately, it should seek a conflict whose difference with the protection set is minimal. This is achieved by finding some minimal set of beliefs B which is inconsistent with the set of protections, and then only branching on B. The algorithm used to accomplish this is described in Section 4.4.3.1. We redefine a node's conflict set to reflect this change:

Definition 10 (Protected Conflict) *For every node n in a protected $\mathcal{HS}$-tree, redefine Conflict(n) to be some minimal set of beliefs in Beliefs(n) such that: Conflict$(n) \cup$ Protects(n) is a conflict.*

By defining conflicts in this way, the protections have already been eliminated from the conflict, so no further checking is required. This also guarantees that protection sets will always be consistent:

Theorem 4 (Consistent Protections) *Let T be a protected $\mathcal{HS}$-tree whose nodes are labeled by conflicts as defined in Definition 10. For every node n in T, Protects(n) is consistent.*

Proof: Assume there is a node n in T such that *Protects*(n) is inconsistent. By Definition 10, *Conflict*$(n) = \emptyset$ and n must be a leaf node. Let p be the parent node of n. *Protects*(p) must be consistent; otherwise p would be a leaf node. Let *new* be the new protections of n relative to p: *new* $\equiv$ [*Protects*(n) − *Protects*(p)]. The set *new* is a proper subset of *Conflict*(p), but the set [*new* $\cup$ *Protects*(p)] = *Protects*(n) is inconsistent. Thus *Conflict*(p) is non-minimal, a contradiction (by Definition 10). □

Definition 4 guarantees that every branch in the protected $\mathcal{HS}$-*tree*yields a hitting set, although not necessarily a minimal one.[7] The consideration of protection sets while searching for conflicts effectively amplifies the pruning potential of the protection sets. Whereas Reiter's algorithm blindly chooses a conflict and then tries to apply the pruning heuristic, this approach informs the generator of the test. By informing the conflict selection process about what actually constitutes a branch in the search, the algorithm takes control of the search and avoids traversing unnecessary paths. Because the reduction in the total size of the $\mathcal{HS}$-*tree*increases exponentially with the depth of the tree, any reduction in branching factor is potentially significant.

4.3.3 Beyond Suspending Component Constraints

In formulating his hitting set algorithm, Reiter was interested in finding minimal retractions of default beliefs about the proper functioning of components. Because the negations of these defaults did not imply anything, there were no minimal conflicts containing any of them. The algorithm as described above finds all minimal conflict-hitting sets only when every minimal conflict is contained in the global set of default beliefs $\mathcal{D}$.

As Section 3.6 describes, this research introduces structural assumptions and other defeasible assumptions whose negations do have consequences. This means that minimal conflicts may contain non-default beliefs and thus not be a subset of $\mathcal{D}$. The algorithm described above does not consider such conflicts, and so is not guaranteed to hit them.

Before trying to repair the algorithm so that it finds all minimal conflict-hitting sets in the general case, we should take a step back and ask ourselves why we are interested in them in the first place. In the simple case of constraint suspension, the minimal conflict-hitting sets correspond to minimal sets of retractions of default beliefs—that is, minimal candidates. With the introduction of behavioral modes and other defaults whose negations have consequences, this correspondence no longer holds. Any conflict-hitting set containing non-default beliefs does not fit our interpretation as a set of beliefs to be retracted from the defaults. One cannot retract what one does not believe.

4.3.3.1 Conflict-Hitting Sets as Negated Beliefs

An alternative view of a conflict-hitting set is as a set of literals to be *disbelieved*, rather than retracted. If we disbelieve every literal in a conflict-hitting set, then our remaining beliefs must be consistent. For some conflict-hitting sets this is not possible to do. Some conflict-hitting sets (even minimal ones) include both a default and its negation; hitting sets containing complementary pairs of literals are called *trivial*, and the remainder are called the *non-trivial* conflict-hitting sets. Trivial conflict-hitting sets ask us to disbelieve a tautology; as such, only the non-trivial conflict-hitting sets are of interest in candidate generation. A simple pruning scheme avoids considering trivial solutions.

The negation of a nontrivial conflict-hitting set is a consistent candidate, though not necessarily a minimal one. De Kleer [24] characterized these candidates as *kernel diagnoses*, and showed that every consistent interpretation is a superset of one or more kernel diagnoses. While every minimal candidate is a kernel diagnosis, not every kernel diagnosis is a minimal candidate.

[7]The use of protection sets eliminates most but not all non-minimal hitting sets; it is still possible to find a non-minimal hitting set down a "left" branch which is a superset of a minimal hitting set found previously down a "right" branch.

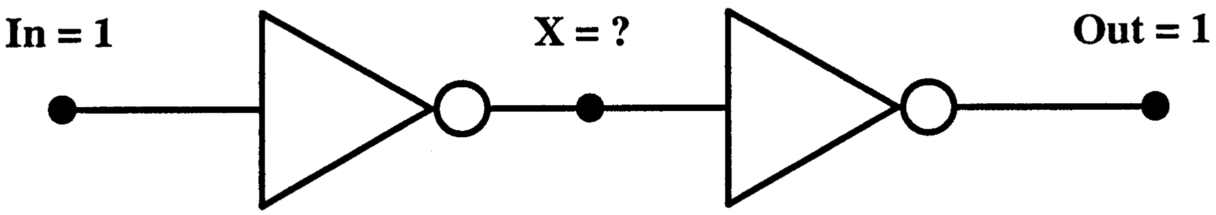


Figure 4.8: Two Inverters in Series

To see this, consider a pair of inverters connected in series as shown in Figure 4.8. Because the observed output matches the predicted output based on the default correctness assumptions, the only minimal candidate indicates that both inverters are working correctly. There are two minimal conflicts for this example: $[A, \neg B]$ and $[\neg A, B]$; and two nontrivial conflict-hitting sets: $\{\neg A, \neg B\}$ and $\{A, B\}$. The first corresponds to the single minimal candidate stating that both inverters are working normally. The second represents a non-minimal candidate where both inverters have failed. Clearly there is no reason to consider the non-minimal candidate as long as the minimal candidate continues to be consistent with observations.

As it stands, the hitting set algorithm will find the first candidate but not the second; that is, it returns the single candidate indicating that nothing is broken, just as we want. The set of default beliefs at the root node is consistent (i.e., it contains no minimal conflict), so the root node is marked with a $\surd$ and the search is terminated. The question is, is there something special about this example, or does the existing algorithm do the right thing in the general case?

Note that the algorithm did not actually find any conflict-hitting sets; it found the empty set $\{\}$, and returned it as a sufficient minimal retraction from the set of defaults. The following theorem demonstrates that the algorithm does find the desired minimal candidates:

Theorem 5 (Minimal Candidates) *Let T be the $\mathcal{HS}$-tree constructed by the hitting set algorithm, after pruning non-minimal hitting sets and protection violations. For every minimal candidate C, there is a leaf node n labeled by $\surd$ such that Retracts$(n) = C$. In addition, for every leaf node n labeled by $\surd$, Retracts(n) represents a minimal candidate.*

Proof: First we show that every minimal candidate is represented by some leaf node in T labeled $\surd$. Suppose there exists a minimal candidate C not represented in T. Because C is minimal, C has no proper subset C' such that $\mathcal{D} - C'$ is consistent. There must exist a node n (possibly the root) in T such that *Retracts*(n) is contained in C, but no child node n_c of n has *Retracts*(n_c) contained in C; otherwise C would be represented in T. This node n is labeled either by $\surd$ or by some minimal conflict.

Case 1: n is labeled by $\surd$: The remaining set of defaults $\mathcal{D}$ – *Retracts*(n) is consistent. But C is a minimal candidate, so *Retracts*(n) cannot be a proper subset of C; thus C = *Retracts*(n), and n represents C, a contradiction.

Case 2: n is labeled by some conflict Σ: Σ must not intersect C, or else n would have a child node whose partial hitting set is in C. But this means that $\Sigma \subset [\mathcal{D} - C]$, which implies that C is not a consistent candidate, a contradiction. Thus every minimal candidate must be represented by some leaf node in T.

Next we show that every leaf node n labeled $\surd$ corresponds to a minimal candidate. A leaf node n is labeled $\surd$ only if $[\mathcal{D} - \mathit{Retracts}(n)]$ is consistent, so $\mathit{Retracts}(n)$ represents a consistent candidate. Because every minimal candidate is represented in T, and by assumption all supersets of the minimal candidates contained in T have been pruned, T contains no non-minimal candidate. Thus for every leaf node n labeled $\surd$, $\mathit{Retracts}(n)$ is a minimal candidate.

Combining the two results above completes the proof. The leaf nodes of T correspond exactly to the set of minimal candidates. □

The above proof demonstrates that the hitting set algorithm finds exactly the set of minimal candidates as desired for diagnosis. There is no point in burdening the candidate generator with finding non-minimal candidates which are always less credible than their minimal counterparts. The non-minimal candidates remain uninteresting as long as the minimal candidates remain consistent with observations.

The hitting set algorithm extends naturally to finding the non-minimal supersets of a minimal candidate. If subsequent observations rule out a minimal candidate C, then there must be a new minimal conflict contained in $[\mathcal{D} - C]$. The leaf node n which represents C must be re-examined to identify the new minimal conflict, and the tree is extended in the normal way.

4.3.3.2 Handling Incompleteness

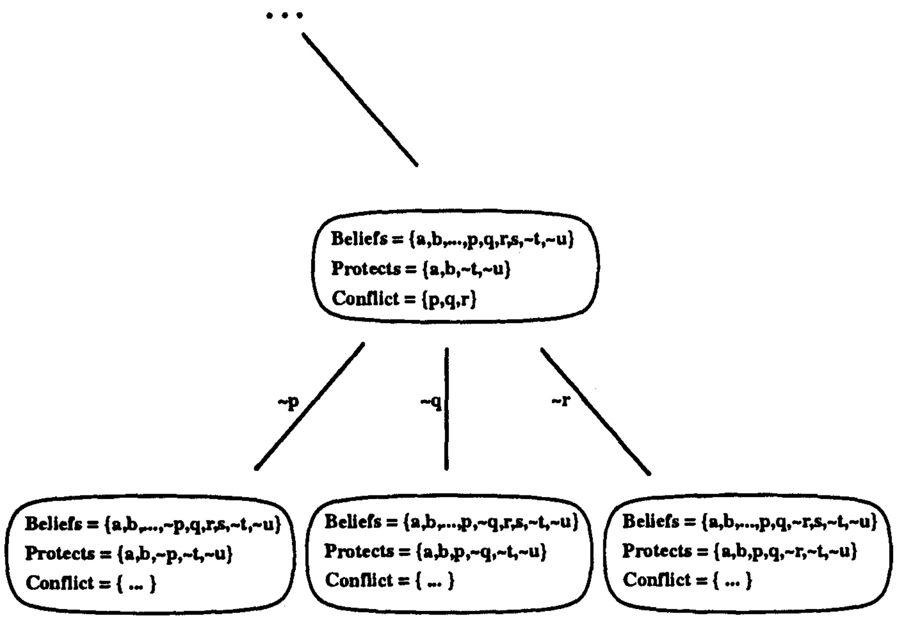
In [68], Reiter assumes the presence of a sound and complete theorem prover to detect conflicts. PDE uses an ATMS as its inference engine, as described in Section 4.4. Because the ATMS is sound but not necessarily complete, it may fail to detect a conflict for some environments. This lack of completeness means that fewer candidates are eliminated, and more effort is expended in the hitting set algorithm.

A sound and complete theorem prover—as assumed by Reiter—would detect as inconsistent any unsatisfiable conjunction of literals. Consider a set of literals Σ and a literal $\sigma \notin \Sigma$, such that $\Sigma \cup \{\sigma\}$ and $\Sigma \cup \{\neg\sigma\}$ are both inconsistent. It follows that Σ is inconsistent as well. Viewed in terms of the hitting set algorithm, let $\Sigma \cup \{\sigma\}$ be an inconsistent belief set for some node. If by removing σ, the resulting set Σ is consistent, then the set $\Sigma \cup \{\neg\sigma\}$ must be consistent as well.

Because the inference mechanism used by the ATMS is incomplete, it is possible for Σ to appear to be consistent even though both $\Sigma \cup \{\sigma\}$ and $\Sigma \cup \{\neg\sigma\}$ are detectably inconsistent. By considering the consistency of $\Sigma \cup \{\neg\sigma\}$ instead of just Σ, it is possible to show that Σ is not consistent after all. In general, the negation of every retracted default may be included in the set of beliefs without affecting the consistency of a minimal candidate:

Proposition 3 (Minimal Negation) *Let Neg() be a function mapping a set of literals to their negations:* $\mathit{Neg}(L) \equiv \{l \mid \neg l \in L\}$. *Let C be a minimal candidate for theory $\mathcal{T}$ and observations OBS. It must be the case that* $[\mathcal{D} - C] \cup \mathit{Neg}(C)$ *is consistent.*

Proof: Suppose C is a minimal candidate, and $[\mathcal{D} - C] \cup \mathit{Neg}(C)$ is inconsistent. By assumption, C is a consistent candidate; that is, $[\mathcal{D} - C]$ is satisfiable. There must be some truth assignment (other than all false) for the literals in C which is consistent with $[\mathcal{D} - C]$. But this truth assignment represents a minimal candidate which subsumes C, so C cannot be a minimal candidate, a contradiction. □



...
Beliefs = {a,b,...,p,q,r,s,~t,~u}
Protects = {a,b,~t,~u}
Conflict = {p,q,r}
~p
~q
~r
Beliefs = {a,b,...,~p,q,r,s,~t,~u}
Protects = {a,b,~p,~t,~u}
Conflict = { ... }
Beliefs = {a,b,...,p,~q,r,s,~t,~u}
Protects = {a,b,p,~q,~t,~u}
Conflict = { ... }
Beliefs = {a,b,...,p,q,~r,s,~t,~u}
Protects = {a,b,p,q,~r,~t,~u}
Conflict = { ... }

Figure 4.9: Negating Belief in the Search for Hitting Sets

The hitting set algorithm may be modified to minimize the potential for incompleteness in the ATMS, by including the negations of the retracted beliefs in the belief set for each node before checking consistency. Thus for each node n in the *HS-tree*, the set of beliefs for n is obtained by starting with the global set of defaults $\mathcal{D}$, removing the retractions *Retracts*(n) and adding in their negations: *Beliefs*(n) = $[\mathcal{D} - \textit{Retracts}(n)] \cup \textit{Neg}(\textit{Retracts}(n))$. By adding the negated retractions into the set of beliefs, conflicts involving these negations may be found and covered.

Consider extending the search from the node depicted in Figure 4.9, whose conflict set is $\{p, q, r\}$. Rather than simply retracting p to resolve the conflict, we negate it by embracing $\neg p$. In order to avoid oscillating on the status of p further down in the tree, we add $\neg p$ to the protection set of the subnode beneath the p-branch. The branches for q and r are handled similarly; the q-branch protects $\neg q$ as well as p, while the r-branch protects $\neg r$ in addition to p and q.

The following theorem and proof demonstrate that including the negated defaults in the belief sets does not alter the hitting sets found, assuming a sound and complete theorem prover:

Theorem 6 (HS Algorithm using Negated Defaults) *Let T be the tree constructed by the hitting set algorithm presented above, but modified to include Neg(Retracts(n)) in Beliefs(n) for each node n. The set of leaf nodes labeled by $\surd$ in T corresponds exactly to the set of minimal candidates.*

Proof: This proof is a generalization of the proof for Theorem 5. First we show that every minimal candidate is represented by some leaf node in T labeled $\surd$. Suppose there exists a minimal candidate C not represented in T. Because C is minimal, C has no proper subset

C' such that $\mathcal{D} - C'$ is consistent. There must exist a node n (possibly the root) in T such that *Retracts*(n) is contained in C, but no child node n_c of n has *Retracts*(n_c) contained in C; otherwise C would be represented in T. This node n is labeled either by $\surd$ or by some minimal conflict.

Case 1: n is labeled by $\surd$: The remaining set of defaults $\mathcal{D}$ − *Retracts*(n) is consistent. But C is a minimal candidate, so *Retracts*(n) cannot be a proper subset of C; thus $C =$ *Retracts*(n), and n represents C, a contradiction.

Case 2: n is labeled by some conflict Σ: Σ must not intersect C, or else n would have a child node whose partial hitting set is in C. But this means that $\Sigma \subset [[\mathcal{D} - C] \cup Neg(C)]$, which by Proposition 3 implies that C is not a minimal candidate, a contradiction. Thus every minimal candidate must be represented by some leaf node in T.

Next we show that every leaf node n labeled $\surd$ corresponds to a minimal candidate. A leaf node n is labeled $\surd$ only if $[[\mathcal{D} - Retracts(n)] \cup Neg(Retracts(n))]$ is consistent. By monotonicity $[\mathcal{D} - Retracts(n)]$ is consistent, so *Retracts*(n) represents a consistent candidate. Because every minimal candidate is represented in T, and by assumption all supersets of the minimal candidates contained in T have been pruned, T contains no non-minimal candidate. Thus for every leaf node n labeled $\surd$, *Retracts*(n) is a minimal candidate.

Combining the two results above completes the proof. The leaf nodes of T correspond exactly to the set of minimal candidates. □

If the theorem prover is not complete, then including the negations of the retracted defaults provides additional information to aid in detecting a conflict. Because one belief is added for each retraction, the cardinality of every node's belief set is the same as that of the root node. If the set of observations includes every state variable, then each belief set represents a complete environment, or *interpretation.* De Kleer [69] proves that the ATMS is *interpretation-complete* with respect to conflicts; that is, an interpretation appears consistent if and only if it is satisfiable. Thus a node n in the $\mathcal{HS}$-*tree* is marked by $\surd$ if and only if *Beliefs*(n) is truly consistent.

If observations only partially specify the model's state variables, it is still possible to verify consistency of a set of beliefs, by searching for some interpretation containing those beliefs. The beliefs are consistent exactly when some interpretation can be found. This process is in some ways analogous to performing physical experiments on the system in order to rule out a candidate; looking for a consistent interpretation may be viewed as a thought experiment. The cost of performing such an experiment and the expectation of success can be computed and used as a basis for dividing resources between various physical and mental experiments.

One other completeness issue involves minimal conflicts. Because the ATMS is incomplete, it may fail to trim down a non-minimal conflict all the way to a minimal conflict. Using non-minimal conflicts in the hitting set algorithm does not affect its correctness, but does introduce additional branches which are barren of any minimal hitting sets.[8] This is the tradeoff of using an incomplete inference engine. The hitting set algorithm is forced to do additional search because the inference engine did not complete the job. Given the extreme cost of a complete proof theory (requiring full clause resolution), this additional branching is typically well worth the cost.

[8]Failure to detect a conflict can affect the correctness of the hitting set algorithm in other ways, by yielding partial hitting sets which appear to be complete. This is especially problematic when the initial conflict between predictions and observations goes undetected.

4.3.4 Incremental Best-First Search

Reiter discusses the inclusion of a depth bound in the hitting set algorithm, to avoid generating hitting sets larger than some fixed size k. For example, single failures are easily found by setting $k = 1$, looking only one level below the root. In general all hitting sets smaller than k would be found and returned as a unit. An alternative is to find and return hitting sets one at a time in best-first order, until some stopping criterion is met.

The approach taken here uses a simple agenda control mechanism to allow early results to be examined before all minimal hitting sets have been found. Candidates are generated best-first, based on some estimate of candidate plausibility. A simple estimate of plausibility is the size of the belief set for each candidate. Using size as the only determiner of plausibility yields breadth-first search, as Reiter proposes. This guarantees that minimal hitting sets are found before considering any non-minimal supersets. Early hitting sets cannot be subsumed by later ones, given that all size–one hitting sets are found before any size–twos, and so on. Fortunately, this same property holds of best-first search, as long as "best" is defined in a reasonable way.

4.3.4.1 Computing Probabilities for Candidates

One way to assign scores to hitting sets is using probability theory. Each assumption may be assigned a probability, based either on subjective experience or on statistical analysis of prior failures. Under the assumption of independence among different failures, the prior probability of a set of component failures is just the product of the individual prior probabilities.

Of course, the independence assumption is frequently violated; failure in one component may propagate to other components, causing them to fail. For example, a shorted resistor may lead to a blown fuse. Assuming independence has the effect of making multiple related failures appear coincidental, and thus under-estimates their probability. Still, the independence assumption allows probabilities to be computed efficiently without requiring complete knowledge of joint probability distributions, and provides a reasonable heuristic for ordering search.

Using prior probabilities and the independence assumption has the desirable property that supersets of negated defaults always have a lower score than their subsets. As with breadth-first search, using probabilities in best-first search ensures that minimal hitting sets are found before considering (and pruning) any non-minimal supersets. If the default beliefs are all assigned the same probability—regardless of the actual value (any value above 0.5)—then the search order reverts to breadth-first.

There are possible exceptions to the minimal-first claim, when posterior probabilities are utilized. It is possible for a non-minimal candidate to explain the observations, while the minimal candidate is merely consistent with them. If the probabilities of the default beliefs are relatively low, or if the observed parameters have many possible values, most of which are eliminated by the non-minimal candidate, then the non-minimal candidate may actually have a higher posterior probability than the minimal candidate it contains. Such an occurrence is considered extremely rare, and is not given further consideration here.

4.3.4.2 Review of the *Min_Retract* Algorithm

A pseudocode version of the algorithm is provided below. The top-level functions an initializing function: *Init_Min_Retract*() and the main function: *Min_Retract*(). *Init_Min_Retract*() is called once to set up the priority queue and construct the root of the tree. *Min_Retract*() may be called multiple times; it returns one minimal hitting set per call, until no more exist, by expanding the next best node on the queue. The function *Expand_HS*() checks for a completed hitting set, and otherwise branches on the beliefs in its conflict, which has already been stripped of protected beliefs. The functions *Consistent*() and *Find_Minimal_Conflict*() are performed by the ATMS, and are described in the next section.

```
Procedure Init_Min_Retract($\mathcal{D}$, $\mathcal{DT}$)
    if Consistent($\mathcal{D}$, $\mathcal{DT}$) then
        return ":CONSISTENT";
    let QUEUE $= \emptyset$;
    let ROOT $=$ build_HS($\emptyset$, $\mathcal{D}$);


Procedure Min_Retract($\mathcal{D}$, $\mathcal{DT}$)
    let hit_set $= \emptyset$;
    repeat
        if QUEUE $= \emptyset$ then
            return ":END";
        hit_set = Expand_HS(Dequeue());
    until hit_set $\neq \emptyset$;
    return Beliefs(hit_set);


Procedure Build_HS(parent, retract)
    let node = Make_HS_Node();
    if parent $= \emptyset$ then        // Building root:
        Beliefs(node) = retract;
        Protects(node) = Retracts(node) $= \emptyset$;
    else Beliefs(node) = [Beliefs(parent) $-$ {retract}] $\cup$ {$\neg$retract};
        Protects(node) = Protections(parent, retract);
        Retracts(node) = Retracts(node) $\cup$ {retract};
    Conflict(node) = Find_Minimal_Conflict(Beliefs(node), Protects(node));
    Enqueue_HS(node);


Procedure Expand_HS(node)
    if Conflict(node) = ":CONSISTENT" then
        return node;
    for each belief $\in$ Conflict(node)
        Build_HS(node, belief);


Procedure Protections(node, retract)
    let protects $= \emptyset$;
    for each belief $\in$ Conflict(node)
        if Index(belief) $<$ Index(retract) then
            add belief to protects;
    return protects;
```

4.4 An Efficient ATMS for Diagnostic Reasoning

The hitting set algorithm described above requires some inference engine to answer queries about conflicts. In particular, the hitting set algorithm provides a set Σ of default beliefs, and requires from the inference engine: a) a determination as to the consistency of Σ; and b) (if Σ is inconsistent) some minimal conflict contained in Σ.

An assumption-based truth maintenance system (ATMS) provides a mechanism for reasoning with multiple situations simultaneously, thereby avoiding duplication of effort inherent in most backtracking schemes. However, the ATMS suffers from a potential exponential explosion of the contexts in which each proposition is true. The incremental nature of the process-based diagnostic approach is inconsistent with the up-front overhead associated with a standard ATMS.

Based on these requirements, a new hybrid ATMS, called CATMS ([5, 29]), was designed and implemented. CATMS avoids most of the cost of a standard ATMS, without sacrificing any (additional) completeness. This section describes CATMS, and demonstrates why it is particularly well-suited to the requirements of candidate generation.

CATMS uses two parallel schemes for avoiding exponential growth. The first involves a redefinition of the concepts of environment subsumption and minimality based on *assumption closures.* The second introduces a focusing scheme to the label propagation algorithm, as suggested in [42]. Each of these is described below.

4.4.1 Theory of Label Compression

The primary responsibility of an ATMS is to process an incremental stream of nodes ($\mathcal{N}$), assumptions ($\mathcal{A}$), and clauses ($\mathcal{C}$) to efficiently answer queries about the status (i.e. true, false, or unknown) of some node in a given environment, or about the consistency of some environment. The label for a node n is defined as the complete set of minimal, consistent environments which together with $\mathcal{C}$ entail n. Any superset environment subsumed by a label environment will also entail n. CATMS generalizes the standard ATMS algorithm by generalizing the notion of environment subsumption. This impacts the minimality and completeness requirements for labels, as explained below.

4.4.1.1 Assumption Closures

In CATMS, an important property of each environment is its *assumption closure*—the set of assumptions which logically follow from its base assumptions:

Definition 11 (Assumption Closure) $\mathit{Closure}(E) \equiv \{A_i \in \mathcal{A} \mid \mathit{Asns}(E) \cup \mathcal{C} \models A_i\}$.

In a standard ATMS, two environments are compared for subsumption by comparing the (base) assumption sets comprising them. In CATMS, subsumption is based on assumption closures:

Definition 12 (Closure Subsumption) E_i c-subsumes $E_j \equiv \mathit{Closure}(E_i) \subseteq \mathit{Closure}(E_j)$.

Closure subsumption is a generalization of standard ATMS subsumption, due to transitivity and the nature of closures:

Proposition 4 (Subsumption Generalized) *If E_i subsumes E_j then E_i c-subsumes E_j; or, if $\mathit{Asns}(E_i) \subseteq \mathit{Asns}(E_j)$ then $\mathit{Closure}(E_i) \subseteq \mathit{Closure}(E_j)$.*

Proposition 4 suggests a test for closure subsumption requiring only a single closure computation:

Proposition 5 (Closure Subsumption Test) *E_i c-subsumes E_j iff $Asns(E_i) \subseteq Closure(E_j)$.*

Finally, there is one additional property of closures—a variation of transitivity—which allows closure subsumption to take the place of subsumption in an ATMS:

Proposition 6 (Subsumption Transitivity) *If E_i c-subsumes E_j and E_j subsumes E_k, then E_i c-subsumes E_k:*
$Closure(E_i) \subseteq Closure(E_j) \wedge Asns(E_j) \subseteq Asns(E_k) \Longrightarrow Closure(E_i) \subseteq Closure(E_k)$.

For a detailed analysis of assumption closures and proofs of the above propositions, see [5, 29].

4.4.1.2 Using Assumption Closures

CATMS transforms the standard ATMS node label by conceptually replacing environments by their assumption closures. Applying closure subsumption to the minimality criterion can eliminate some environments from a node's label; however, it cannot add any, so soundness and consistency are unaffected by the transformation. Thus a node label in CATMS is a subset of the corresponding ATMS label.

By itself, this would violate the completeness requirement for labels; however, label completeness is reestablished by using closure subsumption for queries as well. This follows from Proposition 6. To see this, let E_i and E_j be two environments in *Label*(n) before applying closure subsumption, such that after the transformation E_j is removed as a non-minimal (closure) superset of E_i. Any query environment E_k which was subsumed by E_j will still be c-subsumed by E_i, which remains in *Label*(n). Thus the result of a node query and the completeness of the node label are unaffected by the transformation. See [29] for a formal proof.

4.4.1.3 Label Propagation in CATMS

CATMS computes compressed labels using the standard ATMS label propagation algorithm, but using CATMS's generalized notion of subsumption based on closures. The most dramatic difference occurs when an assumption node is reached during label updating. In CATMS, propagation terminates at that assumption. The CATMS label of an assumption node A always consists of a single environment $\{A\}$. Any other environment that a standard ATMS would include in A's label must necessarily contain A in its assumption closure, and thus be c-subsumed by $\{A\}$. All label propagations through non-assumption nodes proceed as in a standard ATMS, but using the CATMS version of subsumption based on closures.

The blocking of label propagations at assumptions can be understood as follows. In a standard ATMS, label environments are propagated "forward" from the assumed propositions. In CATMS, assumed propositions are, in effect, propagated "backward" to the minimal environments in which they are true and indirectly to the c-subsumed environments, via assumption closures. This avoids the need to propagate through assumptions, thereby reducing the potential for label explosion.

The ability in CATMS to propagate selected nodes (i.e., assumptions) backward to environments provides a continuum between the two extremes of caching nodes with environments and caching environments with nodes. In the extreme case where all propositions are assumed, CATMS performs no label updates, and every label will contain exactly one environment. In that case, CATMS becomes a mechanism for caching the status of each node with particular environments of interest. In the case where no assumptions are justified, the CATMS labels are identical to those of a standard ATMS.

4.4.1.4 Implied Assumptions

The *compression* of CATMS labels accounts for the "C" in CATMS. Label compression can only occur when there are implied assumptions—that is, when some assumption node is contained in some clause. De Kleer has argued [16] that allowing assumptions to be implied makes little intuitive sense. However in practice assumption nodes are commonly implied. This occurs in part because it is often difficult for the problem solver to know in advance which nodes it will eventually assume (as noted in [41]).

The ability to freely include assumptions in clauses without fear of an exponential explosion in the size of node labels (and the time to compute them) removes a great burden from the problem solver. This allows the model builder to focus more on the expressive power of the representations, and less on efficiency issues. We have found this to be a significant advantage.

4.4.1.5 Environment Expansion

The cost of CATMS's compressed labels comes at query time. To answer a query of whether a node N is true in an environment E, CATMS must first *expand* E to the current *Closure*(E), and then check whether E is c-subsumed by any environment in *Label*(n). Because labels in CATMS are potentially much smaller, the added cost of expanding the query environment is offset by the reduced cost of examining a smaller label. Thus the total cost of a query in CATMS may actually be less than in a standard ATMS.

4.4.1.6 Consistency Checking

CATMS labels can be significantly smaller than standard ATMS labels; this is especially true for the label of the contradiction node (which defines the minimal nogoods). Since maintaining consistency is a major expense in an ATMS, CATMS's reduced number of minimal nogoods is an especially significant advantage. As with node status queries, nogood queries require first expanding the query environment to compute its current assumption closure. That closure may be compared against the set of minimal nogoods; if the closure is a superset of a minimal nogood, then the environment is a (non-minimal) nogood. This need to expand is the price that CATMS must pay for having smaller labels and fewer minimal nogoods to check against; empirical results indicate that this tradeoff is in CATMS' favor.

CATMS is able to go one step further to reduce the cost of consistency checking. CATMS does not actually maintain the set of minimal nogoods; the consistency of a new environment is checked in a different way. Each minimal nogood found introduces a *nogood clause*, indicating that at least one of the assumptions in the nogood environment must be false. This allows a non-minimal nogood to infer the negation of some assumptions it contains. Checking consistency of a new environment E is a simple matter of expanding E and checking its closure for a negation pair. This avoids the cost of looking over a potentially exponential set of minimal nogoods for each new environment.

4.4.2 Focused Label Propagation

Even with the efficiency enhancements afforded by label compression, CATMS still spends most of its time in label propagation. Many of the environments which end up in node labels may never be asked about, suggesting that much of this up-front compilation of labels may be unnecessary. One alternative, originally suggested in [42], is to only propagate label environments which are relevant to the current query environment.

Unlike a standard ATMS which reasons with all contexts simultaneously, a focused ATMS limits its reasoning to the task at hand. Rather than computing and propagating labels as soon as clauses and assumptions are built, a focused ATMS postpones label propagation until a query is made. At that time, the query environment becomes the current *focus*, and label environments which are "within focus" are allowed to propagate.

A query in an ATMS takes the form: "Is node n true in environment E?", where n may be the contradiction node, or E may be the empty environment. When a query is made in a focused ATMS, label propagation focuses on the query environment E. Only those environments which could cause n to become true in E are propagated; these are exactly the subsets of E. Thus an environment is considered *within focus* exactly when it is contained in the focus environment; otherwise it is *out of focus.*

4.4.2.1 Blocked Labels

When the very first query is made to a focused ATMS, no label propagation has been performed, so it is obvious where to begin the label propagation process—at the assumptions. Each assumption in the focus environment is queued up for propagation; by definition, every in-focus label environment is derived by propagating these assumptions. For subsequent queries, it is not so obvious where to begin. If the same approach were taken for later queries, existing labels would have to be ignored so that the failure points of the previous propagations could be reached. This would result in a great deal of duplicated effort, since it fails to utilize previous cached results. Every query would pay the same high price as the first. The alternative is to cache the failures explicitly.

Each node n has an associated *blocked label*, designated *Blocked*(n), which lists the blocked environments which have so far failed to propagate beyond n. Whenever the focus changes, the set of blocked label environments is checked against the new focus. Those which are in focus are propagated; the rest remain blocked. The set of blocked environments may be indexed by size and/or contained assumptions, so that only sufficiently small blocked environments need be examined for each new focus.

4.4.2.2 Label Propagation as Marker Passing

It is useful to relate the focused ATMS to a single-context TMS. Label propagation in a simple TMS amounts to marker passing through justifications from antecedents to consequents. A node is marked if it is believed, and is not marked otherwise. If all the antecedents of a justification have been marked, then the consequent is marked as well. The first time a node is marked, it is queued up for consideration of its consequent justifications. Once a node is marked, there is no reason to try to mark it again, since it and all its consequences have been handled or queued to be handled.

In a multiple-context ATMS, a node n may be considered marked with respect to a particular environment E, whenever *Label*(n) contains some subset of E. Just as in the single-context TMS, there is no reason to re-mark a node that is already believed in the current focus. If every antecedent node in a justification is marked by some environment within the focus, then the union of these environments is also within focus, so the consequent will be marked. As long as the focus is consistent, the markers will propagate to reach every node which should be true in the focus environment.

Thus even though an environment E is in focus, it will remain blocked at a node n if n is already known to be true in the focus. This guarantees that node labels grow by at most one environment per query. The only drawback to this approach is that it potentially leaves behind more blocked environments which must be considered for unblocking whenever the focus changes. Empirical results indicate that the reduction in label propagation more than compensates for the expense of the additional blocked environments. If the number of blocked environments ever exceeds the number of nodes, then the straightforward approach of starting from scratch at the assumptions would likely be more efficient, since at most every node is marked once for a given focus.

4.4.2.3 Focusing in CATMS

The use of closures in CATMS introduces a slight complication to the focusing scheme described above. Because environment subsumption in CATMS is based on assumption closures, label propagation must include all environments which fall within the closure of the focus environment. This introduces a kind of bootstrapping problem, because the closure of the query environment is not known until it has been focused upon.

Whenever focused label propagation reaches an assumption, that assumption is added to the closure of the focus environment. Blocked environments which failed to unblock due to being out of focus must be rechecked against the newly expanded focus environment. In some cases this can occur multiple times, requiring the same blocked environment to be checked many times. By waiting until all environments have propagated to completion, the number of subsequent passes over the blocked environments is minimized. Typically only one or two passes are required.

4.4.3 Candidate Generation using CATMS

CATMS was designed and implemented with the specific requirements of candidate generation in mind. Its unique features make it particularly well-suited to this task. In particular, its ability to reason incrementally without having to find all minimal conflicts in advance fits naturally with the hitting set algorithm described in Section 4.3. Also, the ability of assumptions to block label propagation provides a natural mechanism for representing and reasoning hierarchically. These two capabilities are extremely pertinent to candidate generation, and are discussed next.

4.4.3.1 Finding Minimal Conflicts

The hitting set algorithm described in Section 4.3 has been implemented using CATMS as its inference engine. CATMS is given a set of assumptions which may or may not constitute a consistent environment, and is required to make this determination. If the assumptions are inconsistent, CATMS is expected to return a minimal conflict explaining the inconsistency.

The first requirement of this task is just the standard consistency query. CATMS attempts to build the environment and compute its assumption closure. If the environment expands to include some assumption and its negation, then it is inconsistent; otherwise it is consistent.

The second phase involves finding a minimal conflict contained in the set of assumptions. If the assumptions are inconsistent, then CATMS "whittles away" any assumptions which do not contribute to the conflict. Each assumption is individually removed from the set, and the remaining set of assumptions is checked for consistency. If the removal of an assumption eliminates the inconsistency, then that assumption is added back into the set. The whittling continues with the next assumption, until all have been tried. In the end, the remaining assumptions constitute a minimal conflict, since no more assumptions can be removed without eliminating the inconsistency.

As described above, the search for a minimal conflict does not take into account the protection sets of the hitting set algorithm. Recall from 4.3.2.1 that any overlap between the protection set and the minimal conflict found constitutes avoidable search in the $\mathcal{HS}$-*tree*. By informing the conflict minimizing procedure about the protection set, it can avoid trying to retract protected beliefs. If the protection set alone is inconsistent, then all unprotected beliefs will be whittled away, and no branches will result from the protected conflict. The algorithm is given below:

```
Procedure Find_Minimal_Conflict(beliefs, protects)
    if Consistent(beliefs, $\mathcal{DT}$) then
        return ":CONSISTENT";
    for each belief $\in$ [beliefs $-$ protects]
        if $\neg$Consistent([beliefs $-$ {belief}], $\mathcal{DT}$) then
            remove belief from beliefs;
    return [beliefs $-$ protects];
```

This whittling approach does not necessarily find the smallest minimal conflict contained in the original set of assumptions; it merely finds some minimal conflict (given the protected beliefs as true). In general finding the smallest minimal conflict is NP-complete, as shown in [71]. In contrast, the whittling approach considers exactly n consistency queries given n unprotected assumptions, and so is quite efficient. One could exert some additional effort to find a smaller conflict, perhaps by whittling back-to-front in addition to front-to-back, or in some other order. So far the performance of the hitting set algorithm is quite satisfactory using this simple heuristic.

4.4.3.2 Hierarchical Representation

Reasoning often proceeds in a hierarchical manner, starting with the most abstract level and gradually delving into the details. In diagnosis, we may wish to view a particular subsystem as a "black box" while reasoning about the system as a whole. Later as we begin to suspect that the subsystem is misbehaving, we may want to examine its internals in greater detail.

The compressed labels in CATMS provide a mechanism for just such hierarchical reasoning. Consider the example in Figure 4.10. The total system is believed to be functional when all of its major subsystems are operating properly. Each subsystem in turn is functional when all of its components are faultless. This decomposition may be continued to arbitrary depth. By assuming the status of each subassembly, the resulting labels reflect this decomposition. In the

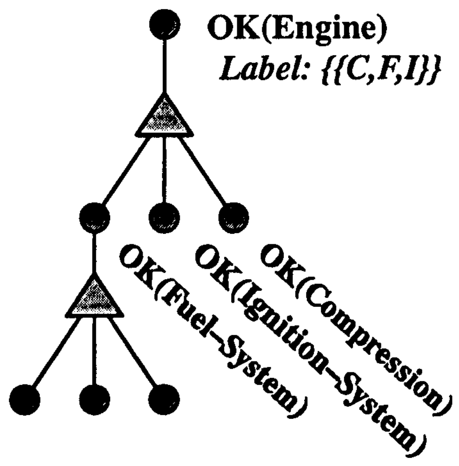


Figure 4.10: Hierarchical Representation of an Internal Combustion Engine

standard ATMS the lowest level details would be propagated all the way up to the system level, swamping the labels with unwanted detail.

Conceptually one could use the justifications themselves to achieve the hierarchy; but justifications do not generally provide good explanations; they are often too fine grained, and do not correspond well with explanatory levels a person might provide. Assumptions in CATMS may be chosen at the appropriate levels to provide the best explanations with the fewest steps to "get to the bottom of things".

In order to ensure that module failures are found instead of individual component failures, we must impose an ordering on the whittling process described above, so that component assumptions are whittled away to reveal conflicts involving module assumptions. Whittling higest probability assumptions first has the desired effect, as the reliability of a module will generally be lower than that of any of its components.[9]

This ability to reason hierarchically is important in diagnosis; for example, if two different modules have failed, there is no reason to consider all the combinations of individual component failures in the two modules. By reasoning at the module level first, we can identify the two general failures, and then focus on each individually. GDE [26] is unable to reason at multiple levels of detail, because it is based on the standard ATMS. Using CATMS, a GDE-like approach can be extended to allow hierarchical representations and the efficiency of reasoning which they afford. Chapter 6 includes examples illustrating hierarchical reasoning using CATMS.

[9]This ignores the use of redundancy, which can lead to module reliability higher than that of its components. In these rare cases, the current approach fails to reason about modules before components. A better ordering based on hierarchical position would correct this.

4.5 Discussion

This chapter has described an approach for diagnosing and repairing an almost-correct model for a broken device. The minimal retraction algorithm presented here improves on Reiter's *DIAGNOSE* algorithm [68], by pruning away branches of the search tree which can only yield redundant explanations. In addition, this algorithm does not just remove suspect beliefs—it negates them, thereby minimizing the potential for lossage due to the incompleteness of the ATMS.

Rymon [70] presents a general algorithm for systematically enumerating a power set, and shows how it may be applied to find hitting sets. In comparing his approach to Reiter's, Rymon interprets Reiter's approach as a two-step process, where the first step identifies the set of conflicts and the second step searches for their hitting sets. This interpretation misses an important point stressed by Reiter regarding his algorithm—namely, that it does not require a pre-enumeration of the set of minimal conflicts. The cost of finding the complete set of minimal conflicts often exceeds the cost of finding their hitting sets. This research shares Reiter's emphasis on the importance of quickly finding the most likely candidates; Rymon's requirement that all minimal conflicts be found in advance is incompatible with the incremental nature of the process-based approach to diagnosis.

The minimal retraction algorithm presented in this chapter is not limited to its current application as a diagnostic candidate generator. Finding minimal retractions to an inconsistent set of beliefs is a common theme in default reasoning. In fact, this minimal retraction algorithm is already being used in IQE to minimize discontinuities following a discrete action, by minimally retracting the predictions derived from continuity to achieve consistency with a known (or presumed) change. This has proven to be much more efficient than the alternative two-step interpretation construction and minimization approach used previously. It is likely that other systems performing temporal or default reasoning could benefit from this algorithm as well.

The candidates generated by the algorithm presented above are not always in the desired structural form. The presence of closed-world assumptions among the set of culpable beliefs results in candidates containing closed-world assumption violations. Such candidates require "fleshing out" to provide a structural description of the failed device. The following chapter presents an abduction algorithm which is used to search for explanations of violated closed-world assumptions.

Chapter 5

Diagnostic Verification Through Focused Abduction

5.1 Introduction

The candidate generator described in Chapter 4 does not always yield a directly testable candidate. In particular, candidates involving closed-world assumption violations do not define the exact structural nature of a failure, or even which components have failed. Candidates involving closed-world assumption violations are thus not acceptable as a final explanation of the symptoms; they merely provide an intermediate representation which facilitates the search for an acceptable candidate—that is, one described in terms of physically observable constructs.

Abduction is the general problem of finding an explanation for an observed or desired condition. In diagnosis, abduction typically involves a search for explanations for a set of observations which conflict with the predictions of a model. Unlike consistency-based diagnosis, which tries to weaken the model until it is consistent with observations, diagnosis using abduction searches for extensions to the model which actually predict the observations.

Given the ambiguity inherent in qualitative reasoning, it is often impossible to construct a qualitative model which precisely predicts a given observed behavior. The predictions of the model are not unsound, they are merely too weak to specify which of several possible behaviors will actually occur. Still, a model which predicts an observation provides a much more plausible explanation than a model whose predictions are merely consistent with the observed behavior.

Rather than trying to find explanations for the observations themselves, the approach taken here is to search for explanations for how a closed-world assumption may have been violated. This is generally simpler and more focused than explaining the observations directly, and still finds a viable structural description; that is, one whose predictions are consistent with observations. Under this approach, a diagnosis is performed in two steps. In the first step, candidates are generated using the minimal retraction algorithm described in the previous chapter. Each candidate defines a model consistent with the given observations, which is obtained by retracting structural, modeling and closed-world assumptions from the original model. In the second step, domain-specific and domain-independent rules are searched abductively for explanations for any violated closed-world assumptions.

This chapter describes an abductive backchaining approach for finding structural explanations for closed-world assumption violations. Given a diagnostic candidate involving a violated closed-world assumption (from Chapter 4), PDE performs an abductive backchaining search for a modified structural description which explains how the closed-world assumption is violated. The abductive algorithm described below is based on the approach of Ng and Mooney [57], but makes important refinements to their approach.

Section 5.2 provides an overview of backward chaining and abduction, including alternative approaches for implementing them. Section 5.3 overviews the ATMS-based Abduction Algorithm (AAA) developed by Ng and Mooney [57], and shows that its limitations make it inapplicable to the abduction task at hand. Section 5.4 presents a Generalized ATMS-based Abduction Algorithm (GA3) which overcomes the limitations of AAA. Section 5.5 describes a focusing technique which orders ATMS label propagation to perform best-first search for abductive explanations. Finally, Section 5.6 summarizes the contributions presented in this chapter.

5.2 Abductive Backchaining

Abductive backchaining is a form of backward chaining search for an explanation of an observed or desired state. This section describes the general nature of backward chaining, with particular focus on abduction.

5.2.1 Forward vs. Backward Chaining

In a forward chaining rule system, input facts are used to trigger rules to compute their consequences. Input facts are typically propositional, and variables in the rules are universally quantified. Forward chaining terminates when every rule whose antecedents match a set of known facts has *fired*, resulting in its consequent being asserted. In backward chaining, rules are run in the opposite direction, matching a given goal with a rule's consequent and subgoaling on the rule's antecedents. Variables in a goal are existentially quantified, indicating that we want to prove the goal for some bindings of its variables. Backward chaining terminates when every rule whose consequent matches a known goal has asserted its antecedents as subgoals.

The direction in which a rule is run is not an inherent property of the rule itself; conceptually any rule may be used for either forward or backward reasoning. The direction used depends on aspects of the domain (such as relative fan-in vs. fan-out, and the finiteness of its deductive closure) and the problem to be solved (such as the number of goals to prove). In qualitative simulation, rules are normally run forward to yield all logical consequences of a given partial state description. Forward chaining is used because we often do not have a single goal to prove, and the cost of computing a state's deductive closure is reasonably small. Backward chaining is important to this research because of its unique ability to reason abductively.

5.2.2 Simple verses Abductive Backchaining

Backward chaining starts with a goal and searches backward through the rules for a set of known or assumable literals which imply it. In simple backchaining the object is to prove that a given goal logically follows from a rule set and a set of already-believed facts. Only true literals constitute successes in the search; no literals may be assumed to be true. If the goal is propositional, then the only issue is whether the goal is deducible from what we know to be true. Any one proof of the goal is no better or worse than any other, and we only need to find one. If the goal has variables which may be bound by individual proofs, then we are interested in the possible bindings for which proofs exist; for any one binding set, a single proof is still as good as a hundred.

By contrast, abductive backchaining searches for possible *explanations* for some observed or desired state of the world. Abductive explanations differ from the proofs of simple backchaining in that they involve assumptions which are not known to be true. Certain forms of literals representing primitive, ground-level relations are designated as "assumable"; these literals may be included in an explanation as if they were known to be true. Each explanation consists of a minimal set of assumptions which if true would allow the goal to be derived.

The presence of assumptions adds an extra dimension to backchaining. In abduction each step may require a different set of assumptions to complete it; some sets of assumptions may be more plausible than others, so all explanations are not created equal.

In practice, abduction amounts to searching for the smallest, most believable set(s) of assumptions whose truth would allow the given goal to be deduced. Because of the similarity between abduction and simple backchaining, abductive explanations are sometimes referred to as proofs, and the phenomena to be explained are referred to as goals (and subgoals). An abductive explanation is indeed a valid proof exactly when all of its assumptions turn out to be true. In the following discussion, the terms *proof*, *explanation* and *derivation* are used somewhat interchangeably to refer to an abductive explanation.

5.2.3 Approaches to Backward Chaining

Two different approaches to backchaining were considered in this research—a *goal-rewrite* approach and an *and-or tree* approach. These differ primarily in the timing of enforcing variable binding agreement. The latter approach won out, and is the basis of the abduction algorithm presented in this chapter.

Suppose we were interested in satisfying the following conjunctive goal: $Politician(?x) \wedge Honest(?x)$. The problem is to find bindings for $?x$ satisfying both predicates. The goal-rewrite approach finds all politicians and then checks each one for honesty. The and/or tree approach independently finds all politicians and all honest persons, and then finds the intersection of the two sets. These two approaches are briefly compared below.

5.2.3.1 Rewriting Goals

The *goal-rewrite* approach to backchaining maintains a queue of alternative "to-do lists" of conjunctive subgoals which entail the original goal. The search begins with a single to-do list containing the original goal. To-do lists are processed one at a time by "rewriting" the first goal on a list, replacing it with a (possibly empty) set of subgoals which entail it. If a goal unifies with a known fact (i.e., a rule with no antecedents), then the result of the rewrite is to eliminate the goal from the to-do list. Each to-do list is rewritten in all possible ways, based on the given rule base, and is then removed from the queue. An empty to-do list indicates success; that is, the original goal has been proven.

In the case of abduction, certain forms of goals are designated as *assumable*, and may be eliminated from a to-do list just as if they were true. Assumable goals require additional bookkeeping, so that when the last goal in a to-do list is eliminated, it is possible to determine the assumptions required to complete the proof, if any. In some contexts it may be necessary to keep track of the rules used and any variable bindings as well. Variable bindings are introduced when a rule consequent unifies with a goal and provides a constant value for one of the goal's variables; bindings are applied to the remaining goals of the rewritten to-do list, so that subsequent rewrites are constrained by the bindings from earlier rewrites.

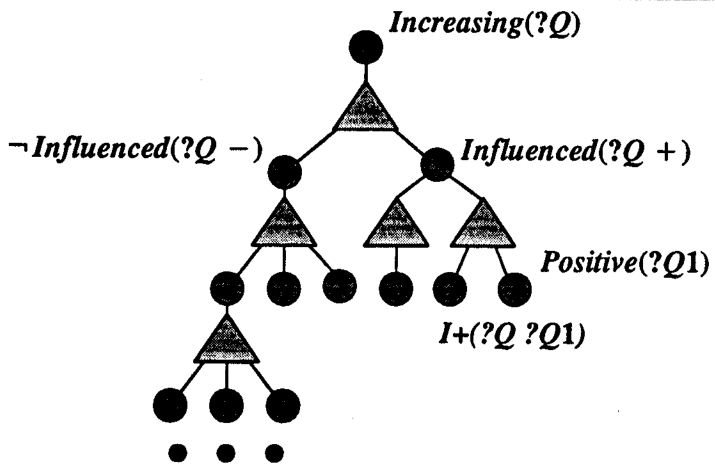


Figure 5.1: A Portion of an And-Or Tree

5.2.3.2 And-Or Trees

Unlike the goal-rewrite approach, backchaining using an *and-or tree*[1] independently finds solutions to conjunctive subgoals, and then checks binding compatibility among the possible conjunctive combinations of proofs. Figure 5.1 illustrates a portion of a typical and-or tree. Each rule instantiation contributes an "and" node representing the conjunction of the rule's antecedents, which are each represented by a single "or" node. The "or" nodes for the antecedents become children of the "and" node, which is a child of the "or" node for the consequent.[2] There may be multiple rules whose consequents unify with the same goal; each such rule provides a different way of proving the goal, each represented by a separate "and" node beneath the goal's "or" node.

Each subtree independently finds proofs for its root goal; these proofs are combined and propagated up the tree. If a goal contains variables, there may be different variable bindings for which the goal is provable, and in the case of abduction, different assumptions required to complete each proof. Thus in the general case each proof must specify both the underlying assumptions and the resulting variable bindings. For each "and" node in the tree, proofs of individual subnodes are combined to yield proofs of the conjunction. The assumptions and bindings from the individual subnode proofs are unioned to yield the assumptions and bindings, respectively, for each proof of the conjunction. In order for the resulting proof to be consistent, its assumptions and bindings must be internally and mutually consistent; in particular, the bindings returned by the individual subnodes are required to agree. If results from two subnodes both bind the same variable, then those bindings must unify. The set of

[1]Because it is possible for the same subgoal to appear in multiple derivations, the and-or "tree" is technically a directed acyclic graph, or *dag*. Cycles are prevented by ignoring branches through nodes occurring on the path to the root node; this is justified because any proof which depends on the goal to be proven is useless. The term "tree" is used here for simplicity.

[2]For uniformity every rule introduces an "and" node even if it only has one antecedent. Thus every node at a given level in the tree is of the same type, and types alternate between "and" and "or" across adjacent levels.

all proofs for the conjunction represents the Cartesian product of the proofs for the individual subnodes, filtered by variable binding consistency.

The leaves of the and-or tree are either true (or assumable) literals or literals which have no rules implying them. The latter represent dead-ends in the backchaining search. Successful proofs propagate up through the tree; if every conjunctive subgoal for some rule succeeds with mutually compatible bindings, then the rule's consequent succeeds with the union of those bindings. Eventually, every possible proof for the original goal will be propagated to the root of the tree.

The decision to use the and-or tree approach in this research was based on the fact that, for the domains considered in this research, subgoals tend to be independent, so that later subgoals are not constrained by the bindings of earlier subgoals. For rules whose antecedent variables are all bound by the consequent, each subgoal is completely independent of the rest. In diagnosis we generally start out with a propositional goal, and most domain rules do not introduce new variables in their antecedents; thus the and-or tree approach is the preferred choice. The following section describes an abduction algorithm which constructs a goal-specific and-or tree of ATMS nodes and justifications, and then uses ATMS label propagation to find minimal sets of assumptions which complete the proof of the goal.

5.3 ATMS-based Abduction Algorithm (AAA)

The ATMS has been described as a general algorithm for performing abduction over a propositional theory [71]. The node labels in an ATMS represent the minimal sets of assumptions which, together with the set of justifications, explain or predict that node's proposition.[3] Each node's label contains sets of assumptions required to explain why that proposition is to be believed. If some proposition is unexpectedly observed to be true, we can look at the corresponding node's label to find possible explanations for why this is so.

Many interesting abductive tasks require the expressive power of a first-order theory. Ng and Mooney [57] describe an ATMS-based Abduction Algorithm (AAA) which extends the abductive ability of the ATMS to first-order Horn theories. AAA achieves significant performance improvements over previous approaches [74].

Basically, the algorithm sets up a tree of ATMS nodes and justifications representing the possible proofs for the top goal. The ATMS nodes at the leaves of the justification tree represent true or assumable literals; the latter are represented by ATMS assumptions. ATMS label propagation yields solutions in the form of environments in the label of the top goal node. Each environment specifies a set of assumable literals which, if all were true, would predict or explain the goal.

This use of an ATMS is somewhat unusual in that it represents first-order formulae instead of just simple propositions. The ATMS does not perform unification or in general support first-order reasoning; this means that it is up to the problem solver to impose the proper constraints among literals. This is what makes this approach interesting.

[3] Justifications may be included in these explanations by adding a unique *justification assumption* as an additional antecedent of each nontrivial justification.

5.3.1 Setting up Justifications

Conceptually AAA constructs an and-or tree for the given goal using ATMS nodes and justifications. Goals are represented by ATMS nodes and rule applications are represented by ATMS justifications.[4] Because a goal may unify with multiple rules, there may be several justifications for a given goal node. Each justification represents an independent proof path for the goal. In effect the justifications represent the "and" nodes of the and-or tree, while the goal nodes represent the "or" nodes.

The algorithm starts by building a node representing the top-level goal. For each newly constructed goal node not already known to be true (or false), the algorithm finds all rules whose consequent unifies with the goal (without binding any variable in the goal; see below). Each such rule results in a new justification of the goal node; the antecedents of the justification are built from the antecedents of the rule, after applying the bindings from the above unification. These antecedents become subgoals and are handled in the same manner. The procedure continues until every node built has been justified in every possible way. New goal nodes are also checked against some assumability criterion; each assumable node is made an ATMS assumption.

The leaves of the resulting justification tree represent either true or assumable subgoals, or subgoals for which no proof is possible. Branches which fail to ground out at true or assumable nodes may be pruned away, along with other branches which depend conjunctively on a pruned branch.

Because this approach requires constructing the complete justification structure for a given goal, it cannot be applied to domains involving infinitely recursive derivations. Fortunately the domains used in this research do not involve recursion, and building the justifications requires only a few seconds. However, laying out the justification tree only sets up the search space; the real work still remains, in the form of label propagation.

5.3.2 Propagating Labels

A subgoal may be recognized as assumable, indicating that it requires no further explanation, and thus provides an acceptable (partial) explanation for the top-level goal. Assumed subgoals are called *ground assumptions* because they define the foundation of the justification structure built by AAA. Ground assumptions containing unbound variables are referred to as *variable ground assumptions*, or *variable assumptions*. The set of variables contained in a variable assumption (or other goal) A_v is denoted by the function $vars(A_v)$. When a new node is built for an assumable subgoal, the node is made an ATMS assumption. The sole purpose in setting up the justification structure is to allow these ground assumptions to propagate to the label of the top-level goal node.

The label of an assumed node contains a single environment representing that assumption in isolation. As labels propagate up through the justification tree, environments from different antecedents of a justification are combined by unioning the assumptions in each. Each environment in the label of a goal node specifies a minimal set of ground assumptions required to prove the goal. By the time label propagation reaches the top-level goal node, the label environments have accumulated the required ground assumptions for every subgoal along some

[4]As explained in Chapter 3, the term *justification* is used to describe what is internally represented in the ATMS as a disjunctive clause.

```
Procedure Abduct($E_i$, CWAs)
    for each cwa in CWAs
        let goal = closed_prop(cwa);
        call Justify_Goal(goal, $E_i$);
    call Prune_Dead_Branches();
    if |CWAs| > 1 then
        goal = a new node;
        justify goal by the closed propositions represented in CWAs;
    repeat
        call Propagate_Best_Label_Env()
    until acceptable explanation reaches Label(goal);
    return Label(goal);
```

Figure 5.2: Top-Level Abduction Algorithm

proof path. Thus standard ATMS label propagation performs exactly the required computation to find minimal explanations for how or why the goal node might be true.

5.3.3 Limitations

Although AAA is reasonably efficient at what it does, the algorithm suffers from a serious limitation: it does not allow variables in a goal to be bound by a rule—it can only prove specific instances given increasingly more general rules. Thus it is not applicable to standard queries of the form: "Is there an $?x$ such that $P(?x)$ is true?".

Even if the top-level goal contains no variables, the same problem may arise for subgoals. It is not uncommon for a rule to introduce a variable among the antecedents which is not mentioned in the consequent. The inability to bind a subgoal variable means that even though a known fact unifies with a subgoal, it cannot be utilized to construct a proof.

By ignoring rules which bind goal variables, AAA completely avoids the issue of when to apply variable bindings. Each subgoal in a conjunction is truly independent of the others, and so may be considered in isolation without any loss of constraint. This explains why the and-or tree which the ATMS naturally implements is the right framework for this task.

Given this inability to prove general goals using specific instances, AAA could not be used as the abductive backchaining component of PDE. The following section describes an algorithm which overcomes this limitation, through the introduction of *binding assumptions*.

5.4 A Generalized ATMS-Based Abduction Algorithm (GA3)

This section presents a general algorithm for performing abductive reasoning in an ATMS, based on the AAA approach. The basic algorithm is given in Figures 5.2–5.4. The algorithm starts by constructing the justifications supporting the goal, just as AAA does. The major difference involves rules which bind goal variables.

Because variables within a goal are existentially quantified, a known proposition which unifies with the goal provides a specialized proof of the goal—that is, an existence proof for a

```
Procedure Justify_Goal(goal, $E_i$)
    if True_In(goal, $E_i$) then
        return;
    if Assumable(goal) then
        assume goal; return;
    for each rule R whose consequent unifies with goal
        let binds = the bindings from above unification;
        let antes = the antecedents of R;
        let $antes_i$ = Apply_Bindings(binds, antes);
        if $\exists ante \in antes_i$ : False_In(ante, $E_i$) then
            do nothing;
        else for each ante $\in antes_i$
                call Justify_Goal(ante, $E_i$);
            for each variable ?v which is free in goal
                    and bound in binds to some val
                let $A_b$ = a new node representing Bind(?v, val);
                assume $A_b$ and add it to $antes_i$;
            build a justification from $antes_i$ to goal;
```

Figure 5.3: Recursive Algorithm for Constructing Justifications

specific instance of the goal. However, if we are to allow AAA to consider such existence proofs, then any environment propagated to the general goal must somehow be tagged as representing a limited proof of a special instance of the goal, rather than an unqualified proof of the original goal. An environment may be specialized by adding an assumption to it, indicating the nature of the specialization. In this case, specializing assumptions serve to distinguish special instances from each other and from the general case. Because these assumptions specify the bindings of certain variables, they are called *binding assumptions.*

5.4.1 Binding Assumptions

A binding assumption is a special ATMS assumption specifying a variable-value pair. Intuitively a binding assumption encodes the belief that a certain variable is bound to a particular value. Because this value is only one of several possible bindings for this variable, the binding is assumed rather than asserted. This allows the ATMS to consider the possibility of a binding without committing to it.

Definition 13 (Binding Assumption) *A* binding assumption *is an* ATMS *assumption representing a variable-value pair. Two functions: var() and val(), map from binding assumptions to their variables and values, respectively: Given a binding assumption A_b having the form: Bind(?v,p), var(A_b) $\equiv$?v and val(A_b) $\equiv$ p.*

Whenever unification of a goal with a rule (or a fact) causes a goal variable to become bound, a binding assumption is added to the set of antecedents for the resulting justification. Such a justification is said to be *specialized* by the binding assumption. Intuitively the binding

```
Procedure Prune_Dead_Branches()
    let Up_Queue = Down_Queue = $\emptyset$;
    for each leaf node node built by Justify_Goal();
        if node is neither assumed nor true in $E_i$ then
            add node to Up_Queue;
    while Up_Queue$\neq\emptyset$;
        let node = Dequeue(Up_Queue);
        for each outgoing justification J from node
            for each ante in antecedents of J
                if J is the only outgoing justification from ante then
                    add ante to Down_Queue;
            if J is the only justification for its consequent C then
                add C to Up_Queue;
            delete J;
    while Down_Queue$\neq\emptyset$;
        let node = Dequeue(Down_Queue);
        for each incoming justification J for node
            for each antecedent ante of J
            if J is the only outgoing justification from ante then
                add ante to Down_Queue;
            delete J;
```

Figure 5.4: Algorithm for Pruning Useless Justifications

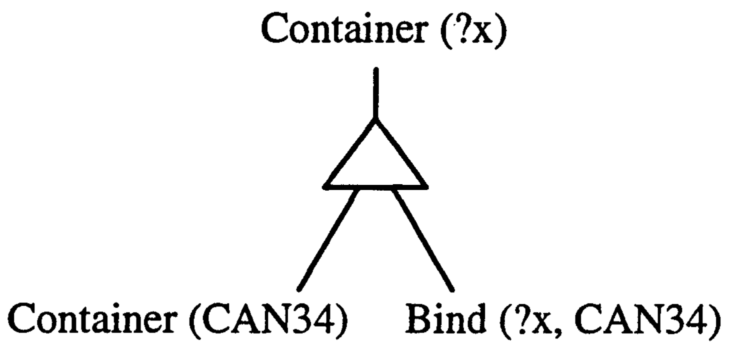


Figure 5.5: An Example Illustrating the Use of Binding Assumptions

assumption indicates that the rule only proves a special instance of the goal. The binding assumption adds its condition to those required by the other antecedents. Figure 5.5 shows an example of how a binding assumption is introduced when a general goal unifies with a specific proposition in the database. This binding assumption may be interpreted as saying: "If you assume $?x$ is bound to CAN34, then *Container*(CAN34) is a proof for *Container*($?x$)". This kind of "proof by existence" is missing in AAA, but enabled by binding assumptions.

Any label environment which results from a specialized justification will include a binding assumption qualifying the environment. As the environment propagates up the justification tree, the binding assumption is carried along, thus indicating the limited scope of any environment derived from this justification. If an environment containing a binding assumption reaches the top goal node, then (at least) a special case of the goal has been explained.

5.4.1.1 Enforcing Unique Bindings

Variables introduced during abduction are existentially quantified, and are typically bound to objects within the model. Each variable may have several possible bindings suggested by known objects within the model. For example, given the goal: *Container*($?x$), and three known containers: $\{CAN1, CAN2, CAN3\}$, the variable $?x$ has at least three possible bindings. In addition, we must allow for unknown objects to fill the role associated with each variable; in the example above, $?x$ could be bound to an unknown container. Every variable $?x$ has a special *skolem binding assumption*, represented as *skolem*($?x$), whose binding object is unspecified. The use of skolem binding assumptions is discussed in Section 5.4.2.2.

Definition 14 (Variable Bindings) *Let $\mathcal{A_B}$ denote the set of all binding assumptions. For each variable $?x$, let bind-asns($?x$) denote the set of all binding assumptions for $?x$: bind-asns*$(?x) \equiv \{A_b \in \mathcal{A_B} \mid \; var(A_b) = ?x\}$.

Definition 15 (Bound Variables) *A variable $?v$ is considered* bound *in an environment E whenever E contains some binding assumption for $?v$: Bound-In*$(?v, E) \equiv [bind\text{-}asns(?v) \cap E] \neq \emptyset$. *The bound value of $?v$ is the value of the binding assumption for $?v$ in E: value-in*$(?v, E) \equiv val(A_b)$, *where* $\{A_b\} = [bind\text{-}asns(?v) \cap E]$.

Each variable can only be bound to a single value. Thus any environment specifying multiple bound values for some variable is a nogood. In particular, any two binding assumptions representing different values for the same variable are mutually inconsistent, and represent a binary nogood. This is enforced by adding a clause for each inconsistent combination:

Proposition 7 (Binding Uniqueness) *Two distinct binding assumptions A_{b1} and A_{b2} are mutually inconsistent whenever they bind the same variable:*

$$\forall A_{b1}, A_{b2} \in \mathcal{A_B}, A_{b1} \neq A_{b2} \wedge var(A_{b1}) = var(A_{b2}) \Longrightarrow [\neg A_{b1} \vee \neg A_{b2}]$$

•

For any two binding assumptions for the same variable, a clause is built indicating that one of the two assumptions must be false. If there are n binding assumptions for the same variable, then there are $\binom{n}{2} = n(n-1)/2$ binary clauses for that variable. These clauses ensure agreement among the bindings suggested by different antecedents of a justification.

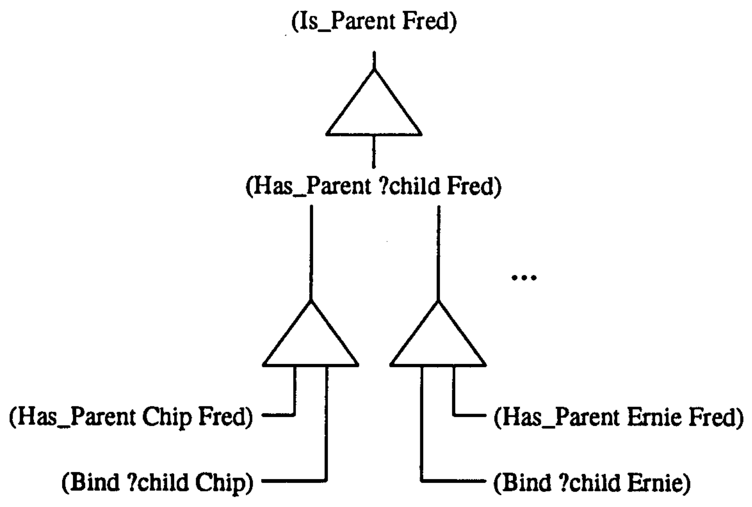


Figure 5.6: Irrelevant Variable Bindings

5.4.1.2 Discarding Irrelevant Binding Assumptions

Abduction can be and often is an extremely expensive endeavor, especially when all possible explanations are sought. The combinatorics of abduction may be decomposed into two components. First, there may be multiple ways of deriving a goal, each having its own distinct general form. Second, within each general form of derivation there may be multiple objects capable of filling a given required role. While either of these components may be of a manageable size, the combination may be intractable.

Certain variable bindings may be of no consequence to the top-level goal, but may be introduced during the proof of intermediate subgoals. If the top-level goal does not mention a particular variable ?v, but some subgoal does, then multiple proofs differing only in the binding of ?v are redundant. Figure 5.6 shows the justifications supporting the top-level goal to show that Fred is a parent: `(Is_Parent FRED)`. Backchaining yields the subgoal: `(Has_Parent ?X FRED)`. Fred may have fathered several children, each of which suggests a binding for ?x. But the original query concerns Fred's parental status—not the names of his children. The problem is compounded when multiple subgoals all return multiple irrelevant answers. For example, given the query: "Is Fred both a parent and a pet owner?", we could find a separate answer for each combination of child and pet in Fred's home. The query contains no variables, so variable bindings cannot possibly be relevant to its answer. For efficiency, we would like to avoid considering all combinations of such bindings. Still, bindings must be considered at least long enough to ensure that the same object fills a given role along each branch of a derivation. Here we discuss a technique for avoiding this combinatorial explosion of variable bindings.

When backchaining on a goal, a rule may introduce new variables which appear in its antecedents but not its consequent. These are called the *terminal* variables of the rule or its resulting justification:

Definition 16 (Terminal Variables) *A* terminal variable *$?v$ for a justification J is a variable mentioned by some antecedent of J but not by the consequent of J. J is called the* terminating justification *for $?v$.*

Given a justification J for a goal G, each terminal variable for J will only appear in the subtree rooted at G. Label propagation may yield environments in the label of G containing binding assumptions for these terminal variables. The terminal variables of J are not mentioned in G; thus G is not constrained by their bound values. The binding of a terminal variable is said to be *irrelevant* to G:

Definition 17 (Variable Relevance) *The bindings for a variable $?v$ are considered* relevant *to a goal G whenever G mentions $?v$: $?v \in vars(G) \Longrightarrow Relevant(?v, G)$. Similarly, the bindings for $?v$ are relevant to an environment E whenever E contains a variable assumption mentioning $?v$: $[\exists A_v \in E : ?v \in vars(A_v)] \Longrightarrow Relevant(?v, E)$. In all other cases, the bindings of $?v$ are* irrelevant.

A variable is irrelevant to any goal in the justification tree above its terminating justification. These higher goals do not mention the variable, and so are not constrained by its binding. When a label environment E containing a binding assumption A_b propagates through the terminating justification for A_b, then A_b has completed its primary mission of specializing goals. However, E may contain a variable assumption containing the free variable $var(A_b)$, indicating that the environment itself is constrained by the binding. If A_b is irrelevant to both G and E, then A_b may be dropped from E in G's label.

Like de Kleer's ATMS, CATMS provides various hooks for user-defined procedures to be applied when specified internal events occur. De Kleer calls these *consumers*; in this discussion the term *handlers* is used. In particular there are three types of handlers corresponding to three types of ATMS events. A *node handler* is associated with a particular node, and is applied to each new environment added to that node's label. An *environment handler* is applied to each new environment as it is constructed. A *nogood handler* (called a *contradiction rule* in [42]) is applied to each environment as it becomes contradictory. These handlers are utilized in various ways throughout the abduction process, as discussed later in this chapter.

Whenever a rule includes terminal variables, the justification built for that rule is redirected to a special node called an *envoy* for the actual consequent goal node. The envoy is a copy of the goal node which receives the label environments from this justification only. A node handler links the envoy with the actual goal node. The primary function of the node handler is to filter out irrelevant binding assumptions from environments produced by the justification, before passing them on to the actual consequent node.[5] For each environment E propagated to the envoy, the handler searches E for irrelevant binding assumptions for its terminal variables. If any are found, the handler builds a new environment E' which is identical to E but with irrelevant binding assumptions removed, and inserts E' into the label of the consequent node. Environments containing no irrelevant binding assumptions are passed unaltered to the consequent node's label. The algorithm for filtering irrelevant binding assumptions is presented below:

[5]Section 5.4.2.2 describes how this same node handler is used to introduce skolem constants.

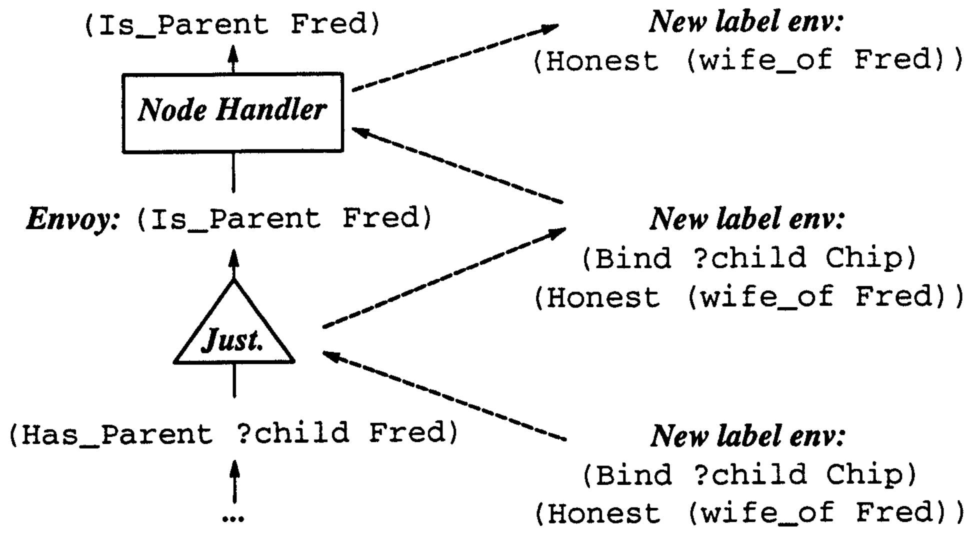


Figure 5.7: Filtering Redundant Binding Assumptions

```
Procedure Handle-Envoy1($E_i$, Goal, T_Vars)
    // Called when $E_i$ is added to Label(Envoy)
    let Irrelevants = $E_j$ = $\emptyset$;
    for each terminal variable ?v $\in$ T_Vars
        if $\neg$Relevant(?v, $E_i$) then
            add ?v to Irrelevants;
    for each $A_i \in E_i$
        if $A_i$ is not a binding assumption or var($A_i$) $\notin$ Irrelevants then
            add $A_i$ to $E_j$;
    Construct environment $E_j$ and add it to Label(G).
```

Figure 5.7 shows how the node handler is applied to the example in Figure 5.6. Here the binding of *?child* is irrelevant to the top-level goal; it only matters that some *?child* is found. The node handler strips out the irrelevant binding for *?child* before propagating the result to the consequent node. By filtering out irrelevant binding assumptions, multiple possible bindings for subgoal variables do not contribute to a combinatorial explosion of labels for goals higher up in the justification tree.

5.4.2 Forward Chaining on Variable Assumptions

The justification tree built by AAA is only a portion of a larger forest of potential justifications which are sanctioned by the rule set. Primitive and intermediate level subgoals may have logical consequences not captured in the tree. Failure to consider the consequences from the missing justifications can waste time on inconsistent environments. These consequences may be considered by triggering and running rules in the forward direction, given the intermediate nodes in the tree.

Forward chaining performs two roles in the abduction process. First, it provides a reality check to ensure that only consistent explanations are considered. Second, it provides additional

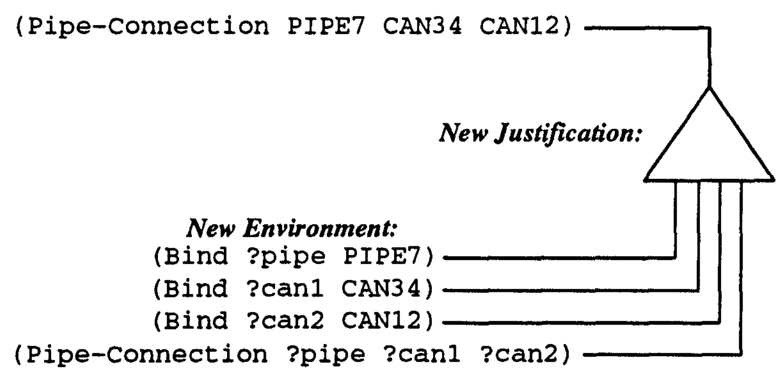


Figure 5.8: Instantiating a Variable Assumption

information regarding what must be true given the current hypothesis, which can be used in evaluating its plausibility.

The forward chaining rules in ARM do not support variables in the fact database, and so can only be applied to propositions. In particular, variable assumptions are not considered for forward chaining. In order to consider the consequences of a variable assumption, it must first be instantiated by replacing all variables by their bound values.

5.4.2.1 Applying Variable Bindings

In order to forward-chain on partial results, bindings are spliced in to instantiate variable forms. Each variable has a set of possible values being considered so far, in the form of binding assumptions for that variable. Rather than considering all combinations of bindings in advance, the approach taken here is to only instantiate a variable form only when some environment includes a complete set of bindings for it.

Definition 18 (Variable Assumption Instantiation) *Consider an environment E containing a variable assumption A_v. A_v is said to be* fully instantiated *in E whenever all of the variables in A_v have binding assumptions in E:*
Instantiates$(E, A_v) \equiv [\forall ?v \in vars(A_v), Bound\text{-}In(?v, E)]$. *The* instantiated form *of A_v is obtained by replacing each variable $?v_i$ in A_v with its bound value, value-in$(?v_i, E)$.*

Whenever a new environment contains a variable assumption and enough binding assumptions to fully instantiate it, the instantiated form of the variable assumption is justified by these assumptions. Figure 5.8 shows an environment containing a fully-instantiated variable assumption. The instantiated form is justified by the variable assumption and the relevant binding assumptions. This is implemented using an environment handler, which checks each new environment E for fully-instantiated variable assumptions (given the binding assumptions in E). For each variable assumption A_v whose variables are all bound in E, the handler ensures that the instantiated form F of A_v is true in E. If F is not already believed in E, the handler justifies F by A_v and the relevant binding assumptions. Note that this is only done once for each set of bindings: if a later environment is built which also contains these assumptions, then n_f will be known true, and no justification will be built.

Because F contains no variables, it is a legal proposition for forward chaining. Forward chaining on F allows reasoning about its other consequences and ultimately its consistency, both in isolation and in combination with other facts.

5.4.2.2 Skolemizing Unbound Variables

Sometimes it is desirable to name an unknown object represented by an unbound variable, to enable forward reasoning. Consider the terminating justification J for some variable $?v$; call its consequent G. Because $?v$ is not mentioned outside the subtree rooted at G, it is not possible for propagation above G to contribute a binding for $?v$; thus J represents the last opportunity for v to become bound. If a label environment E reaches G without specifying a binding for $?v$, then $?v$ will remain unbound at all nodes above G in the tree. The fact that $?v$ is unbound in E indicates that a particular binding was not required to complete the proof for G; any binding will suffice.

If E contains some variable assumption A_v which mentions $?v$ (and no other unbound variables), then the lack of a binding for $?v$ prevents us from instantiating $A_v$and reasoning about its implications. In order to forward reason on A_v, we introduce a *Skolem* binding for $?v$, indicating that there exists some object filling the role of $?v$. For each variable $?v$, there is one Skolem binding assumption denoted *skolem*$(?v)$, which provides a default binding to be used when no other binding is specified.

The node handler introduced in Section 5.4.1.2 is augmented to perform the additional task of ensuring that all relevant terminal variables are bound. For each environment E_i propagated through J, the node handler identifies relevant terminal variables which are unbound in E_i. In addition to stripping out irrelevant binding assumptions, the node handler adds in the skolem binding assumption for each unbound, relevant terminal variable. The resulting environment is propagated to the label of G as described in Section 5.4.1.2. The algorithm for the node handler is shown below:

```
Procedure Handle-Envoy($E_i$, Goal, T_Vars)
    // Called when $E_i$ is added to Label(Envoy)
    let Irrelevants = $E_j$ = $\emptyset$;
    for each terminal variable $?v \in$ T_Vars
        if Relevant($?v, E_i$) then
            if $\neg$Bound-In($?v, E_i$) then add skolem($?v$) to $E_j$;
        else add $?v$ to Irrelevants;
    for each $A_i \in E_i$
        if $A_i$ is not a binding assumption or var($A_i$) $\notin$ Irrelevants then
            add $A_i$ to $E_j$;
    Construct environment $E_j$ and add it to Label(G).
```

The fact that no antecedent provides a binding for a terminal variable means that any binding will suffice to prove the goal. Skolemizing the unbound terminal variables allows forward reasoning on the rules without requiring that rules explicitly handle existentially quantified variables.

5.4.3 Avoiding Redundancy

The other significant advance of this approach over AAA involves the caching of subgoals to avoid redundant proofs. The same subgoal may show up in different parts of the justification

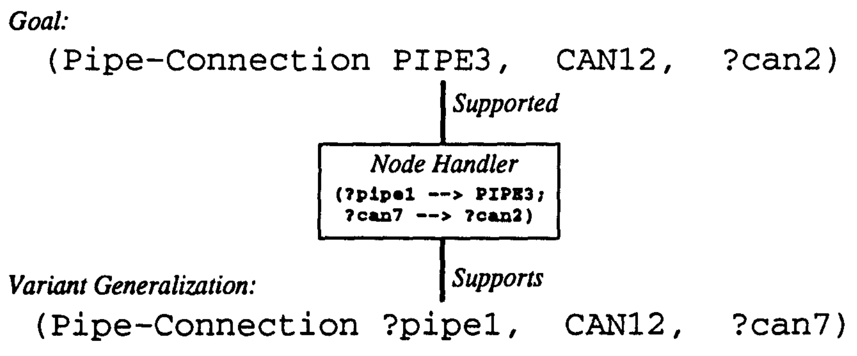


Figure 5.9: Reusing Existing Goals

tree, possibly with different variable names but otherwise identical. A great deal of wasted effort can result from proving the same goal multiple times. This section describes the reuse of existing justification subtrees for avoiding redundancy in the abductive search.

5.4.3.1 Reusing Structure

It is not uncommon for two identical or similar subgoals to show up in different portions of the justification tree. Two subgoals often differ only in the names of their variables, but are otherwise identical. These are called *variants*, and occur due to the need for unique variable names for each application of a rule. Two variant subgoals may or may not refer to the same entities, and so must be treated as distinct.

While AAA does not build multiple nodes for a specific goal, it does build separate nodes for variant subgoals. The proofs for each variant will all be the same except for the names of the variables used. These proofs can involve large portions of the total justification tree and require a significant computation during label propagation.

These redundant proofs can be avoided by caching the results for a single variant, and transforming these results (to reflect the variable renaming) for the other variants. Only one variant—called the *supporting* variant—is justified in the tree; the remaining variants are *supported* by the supporting variant.

In general, any two goals which are unifiable can potentially provide proofs for each other. However, in order to be guaranteed of providing all possible proofs, the unification must not bind any variables in the supported goal. In such cases, the supporting goal is called a *variant generalization* of the supported goal.

Figure 5.9 shows an example of a specific goal being supported by a variant generalization. A variant generalization of a specific subgoal must find (at least) all the proofs that the specific subgoal would find; the specific subgoal need only select those proofs which are relevant to it, and reinterpret them in terms of its own variable names.

5.4.3.2 Translating Solutions

For each new goal added to the justification tree, GA3 checks for an existing variant generalization to support the new goal. If one is found, then no backchaining is performed on the new

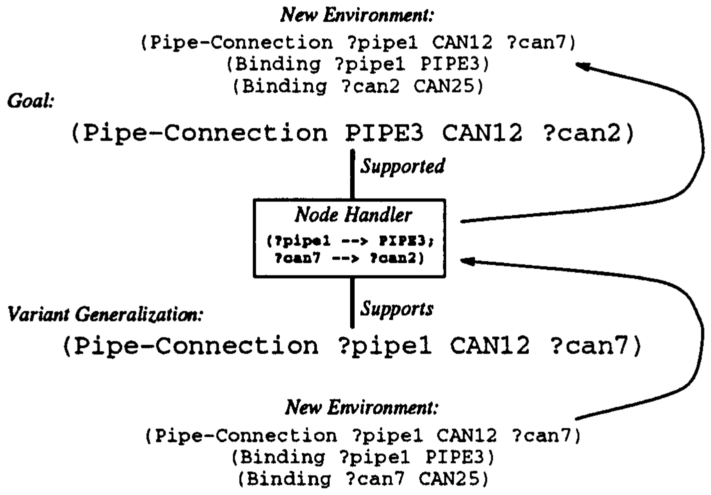


Figure 5.10: Translating Proofs for Specific Variants of a General Goal

goal. Instead, the two goal nodes are linked to reflect the support to be provided by the variant generalization. The link consists of a node handler attached to the supporting node. The node handler fires each time a new label environment E propagates to the supporting node; its job is to transform E and pass the result to the supported node. The transformation reflects any variable renaming and/or specialization required by the supported goal.

The handler caches the bindings from the original unification between the supported and supporting goals; these bindings are applied to the variable assumptions and binding assumptions of supported label environment E yielding a new environment E'. If E' is consistent, then it is added to the label of the supported node. Figure 5.10 shows an example of a label environment being transformed and passed across a support link.

This "piggybacking" of variant goals can significantly reduce the size of the justification structure and the resulting complexity of abduction. Conceivably one could do even better by finding general proofs which can support multiple goals having the same predicate. Alternatively, one could generalize subtrees of justifications using EBL techniques, replacing certain constants by variables. This would potentially create more variant generalizations to provide proofs for otherwise unsupported subgoals. There are many open research issues here; finding the optimal justification tree would likely involve more work than it would save. The simple approach taken here avoids the obvious redundancies, and seems like a reasonable cutoff.

5.5 Focusing on Simple Explanations

As presented so far, GA3 finds all explanations for a given goal. In the general case abduction is known to be exponential, so it is impractical to find all explanations. This section describes an approach for ordering the search for explanations, based on estimates of explanatory coherence.

5.5.1 Evaluating Partial Explanations

Some explanations are more reasonable than others, and some partial explanations are closer to being complete than others. These simple facts provide the basis for ordering the search for abductive explanations. Best-first search requires ordering alternative frontiers of the search, based on estimates of the eventual cost of each partial path. One measure of the cost of an abductive explanation is its a priori probability of being incorrect. This in turn is a function of the assumptions required to complete the proof of the goal.

Each assumption is assigned a probability, indicating the relative likelihood that that assumption participates in a correct explanation. In the simple case where all ground assumptions are considered equally probable, minimizing probability reduces to minimizing the number of ground assumptions required to complete a proof. In addition, we would also prefer to minimize the number of unknown objects required by the proof. Known objects are preferred over unknown objects to fill the various roles within the explanation. This is accomplished by assigning low probabilities to skolem binding assumptions relative to the other binding assumptions for the same variable. Given an estimated probability for each assumption, and assuming independence among the different assumptions, the probability of an explanation is computed as the product of the individual probabilities for the assumptions it contains.

In order to avoid expensive label propagation, we need to evaluate label environments before they reach the top-level goal node, to decide which ones to propagate next. This requires estimating the eventual cost of each potential explanation. Label environments for intermediate subgoals specify sets of assumptions required so far, and further propagation can only add more required assumptions. Thus estimated eventual cost may be decomposed into a sum of the actual cost so far and an estimate of the cost remaining. The remaining cost for a label environment propagating up through the justification tree is not known in advance, but is determined by the portion of the tree remaining to be climbed. One estimator of this remaining cost is the fraction of the total proof remaining to be solved.

Each subgoal is assigned a *proof fraction* indicating the fraction of the overall problem it solves. The top-level goal is assigned a proof fraction of unity, and the proof fractions of lower subgoals are computed by dividing credit evenly among conjunctive subgoals.

Each justification divides the proof fraction of its consequent evenly among its antecedents. For example, if a subgoal G has a proof fraction of $1/4$, and a justification J for G has three antecedents, then each antecedent of J receives a proof fraction of $1/12$. Proof fractions continue to shrink as they are divided up among the multiple antecedents of each justification, until some small proof fraction reaches the leaves of the tree.

Label environments are also assigned a proof fraction, defined as the maximum proof fraction of all the subgoal nodes reached so far. The estimated total cost for a partial proof is computed assuming that the rest of the proof will be just as difficult as the portion completed so far. Thus the estimated probability for a partial proof is computed as the probability so far, raised to the power of the inverse of the proof fraction. For example, suppose a label environment E contains ground assumptions having a combined probability of p_e; if the proof fraction for E is $1/n$, then the estimated probability of a complete proof built from E is $(p_e)^n$.

This heuristic has the effect of favoring label environments higher in the justification tree. Environments which still look promising after propagating most of the way up the tree are encouraged to continue their climb; thus "the cream rises to the top" more quickly and with less overall propagation at the lower levels in the tree.

5.5.2 Best-First Label Propagation

The motivation for assigning costs to environments is to order the propagation of label environments so that simple, coherent explanations are found with the least possible effort. To the extent that the cost estimates are reasonable, label environments for intermediate subgoal nodes with the lowest costs are most likely to yield low-cost explanations in the label of the top-level goal node.

The cost estimates for label environments are used to order label propagation to consider best explanations first. A priority queue is maintained for blocked label environments, ordered lowest cost first. Until the queue is empty, the lowest cost blocked environment is unblocked, allowing it to propagate further up the justification tree to combine with other label environments. Each new label environment is automatically blocked and inserted into the priority queue.

Whenever an environment propagates to the top-level goal node, all label propagation ceases in order to allow the new explanation to be evaluated. Further propagation may be continued to ensure that this is indeed the best available explanation. This is guaranteed when every blocked label environment has a higher cost so far (based on its ground assumptions only, ignoring proof fraction) than the actual cost of the new explanation.

If an explanation is later discredited, the next best explanation is found by continuing the unblocking process, until a suitable explanation is found.

5.6 Discussion

This chapter has presented a general abduction algorithm which is used to explain violated closed-world assumptions. The candidate generator from the previous chapter provides both the goal to be proven and the environment in which to focus the search. This two-stage process provides an effective decomposition of the overall problem, resulting in much more efficient search as compared with a blind search for explanations of the observations themselves. Also, the ambiguity inherent in qualitative models makes explaining a missing influence or other closed-world assumption violation much more likely to succeed than explaining the observations directly.

The GA3 abduction algorithm presented in this chapter uses ATMS label propagation to compute abductive explanations for a given goal. This approach extends Ng and Mooney's AAA [57] to allow goal variables to be bound during unification with a rule—an essential requirement for diagnostic verification in PDE. Binding assumptions are introduced to allow specific instances to provide limited "existence proofs" of a general goal. A special mechanism for removing irrelevant binding assumptions which have completed their mission avoids a combinatoric explosion of irrelevant bindings being propagated to the top-level goal. Instantiation of variable assumptions allows forward chaining in cases where all included variables are bound. The use of skolem bindings provides a mechanism for naming unknown objects represented by unbound variables; this extends the potential for forward reasoning to unknown objects.

A system called FORLOG [36, 52] is similar to GA3 in its use of binding assumptions and Skolem constants; FORLOG uses an ATMS and its *consumer architecture* to implement a logic programming language similar to PROLOG. It was not specifically designed for abductive reasoning; still, the similarities are worth noting.

Abduction is inherently exponential. Some might view the search for an efficient approach to abduction as a waste of time pursuing an impossible goal. However, given the importance of abductive reasoning in this and other research, together with its prominence in the total cost of problem solving, the abduction algorithm presents the greatest opportunity for overall performance improvement. This chapter has presented two significant performance improvements over Ng and Mooney's AAA. The first is the use of variant subgoals to avoid redundant structure. The second is the application of best-first search to label propagation, based on an estimated final probability of success. Together these two modifications contribute significantly to the overall performance of the abduction algorithm. The following chapter illustrates the performance of the abduction algorithm as well as the candidate generator, through several implemented examples.

Chapter 6

Examples

6.1 Introduction

This chapter describes several examples of failures diagnosed under process-based diagnosis and its implementation, PDE. These examples are intended to serve three primary roles. First, they help to elucidate the ideas presented in this thesis. Many of the examples are deliberately simple to illustrate the fundamentals of the approach, rather than the limits of its capabilities. Some of the examples were conceived before the approach itself was fully developed, and have provided much of the direction in its evolution. Second, the examples serve to substantiate the claims of the thesis; in particular, the claim that process-based diagnosis is capable of explaining failures where other approaches are either too weak or too inefficient to be applied. Third, these examples demonstrate the performance of the implementation of the approach, PDE. Each example includes run times[1] and other performance statistics which indicate the complexity of the example and the efficiency of the approach and its implementation.

The diagnostic candidates and abductive explanations produced by PDE consist of ATMS environments specifying consistent sets of assumptions which could explain the given symptoms. These environments can be quite large, and so are represented concisely by their differences from the original set of base beliefs. Also for conciseness, the results given in this chapter exclude candidates in which the observations themselves have been retracted; it is always possible to explain a failure by questioning the observations leading to the initial conflict. Observation failures (Section 3.6.6) are excluded by assigning a probability of 0.9999 to the observation assumptions, so that other candidates are found first. Other default assumptions are given lower probabilities; for example, closed-world assumptions are given 0.999; scenario assumptions are given 0.99; and modeling assumptions have 0.9 probabilities. A more refined approach would assign each probability according to specific failure rate information; the numbers used here are based on the intuitive ordering of the different types of failures, and are adequate to produce candidates in a reasonable order.

The first two examples in this chapter involve simple electronic circuits, which are familiar to model-based diagnosis researchers. Section 6.2 evaluates the performance of the candidate generator using a multiple-bit ripple-carry adder, originally used in [20] to compare the performance of GDE and *Sherlock*. Section 6.3 describes a circuit consisting of a battery and two

[1]Run-times for all examples are for a Sun Microsystems SPARC II with 48 megabytes of memory.

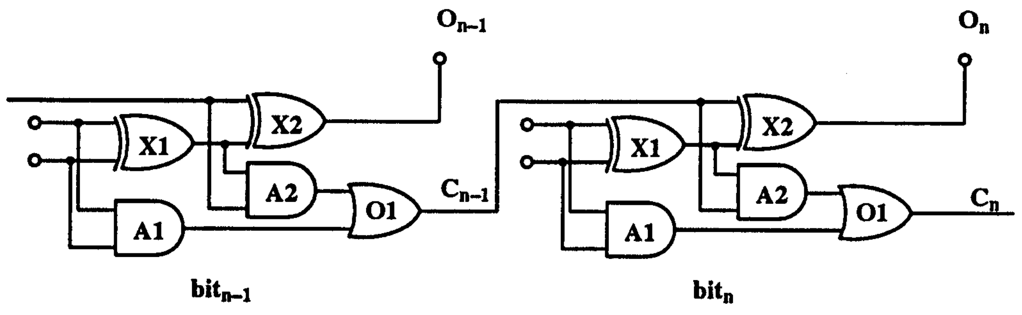


Figure 6.1: Two Bits of an N-Bit Adder

light bulbs, one of which has failed. We use this example to demonstrate how a closed domain theory can perform the same function as explicit failure modes.

The remaining examples are drawn from the domain of thermodynamics, and utilize the domain theory presented in Appendix A. Among the domains modeled using QP theory, thermodynamics has from the beginning been a primary focus of model development research within the Qualitative Reasoning Group. Section 6.4 describes a simple two-container fluid system in which the level of liquid in one container is unexpectedly falling. Section 6.5 describes a different set of observations for the two-container fluid system, in which a pressure difference fails to change the levels of liquid in the two containers. Section 6.6 is based on the same scenario as above, but where the liquid levels are changing more slowly than expected. Section 6.7 describes a more complicated fluid system, in which a leaky valve causes a liquid level to unexpectedly rise. Section 6.8 demonstrates performance as the size of the fluid system increases, using totally-connected and Manhattan grid configurations of containers.

6.2 An N-Bit Adder

Figure 6.1 shows two bits of a larger n-bit adder. Each bit consists of five components: two and-gates, two xor-gates, and one or-gate. Each bit has three inputs (the two bits to be added and a carry bit), and two outputs (a sum and a carry). The carry output from each bit connects to the carry input of the next (more significant) bit.

This example differs from the others in this chapter in that it is not based on a causal, process-centered model, and so does not involve the abduction component of PDE (there are no closed-world assumptions to violate). It is taken from [20], and is included to demonstrate the generality and performance of the minimal retraction algorithm at the heart of the candidate generator. In addition, this example is used to demonstrate the benefits of a hierarchical representation, by reasoning about the adder at the bit-level.

Figure 6.2 lists the rules used to model logic gates. Inputs and outputs are either high or low. Each type of gate enforces the appropriate logical constraint between its inputs and its output, using the logical relations defined in Section 3.4. Each component is assumed to have only two failure modes: `stuck-at-0` and `stuck-at-1`. Each failure mode is given a probability of 0.01.

```
;;;; Rules for Logic Gates

(=TR=> (Low ?x) (not (High ?x)))
(=TR=> (OK ?gate) (not (or (Stuck-At-Zero ?gate) (Stuck-At-One ?gate))))

(==> (Gate ?type ?gate ?output . ?inputs)
     (and (==> (Stuck-At-One  ?gate) (High ?output))
          (==> (Stuck-At-Zero ?gate) (Low  ?output))))

(==> (Gate INV ?gate ?out ?in)
     (==> (OK ?gate) (iff (High ?out) (Low ?in))))

(==> (Gate AND ?gate ?out ?in1 ?in2)
     (==> (OK ?gate) (iff (High ?out) (and (High ?in1) (High ?in2)))))

(==> (Gate IOR ?gate ?out ?in1 ?in2)
     (==> (OK ?gate) (iff (High ?out) (or  (High ?in1) (High ?in2)))))

(==> (Gate XOR ?gate ?out ?in1 ?in2)
     (==> (OK ?gate) (iff (High ?out) (xor (High ?in1) (High ?in2)))))
```

Figure 6.2: Rules for Logic Gates

```
;;;;  TYPE   NAME     OUTPUT       INPUT1       INPUT2

(Gate  XOR  (X1 1)  (x1-out 1)  (in1 1)     (in2 1))
(Gate  XOR  (X2 1)  (out 1)     (carry 0)   (x1-out 1))
(Gate  AND  (A1 1)  (a1-out 1)  (in1 1)     (in2 1))
(Gate  AND  (A2 1)  (a2-out 1)  (x1-out 1)  (carry 0))
(Gate  IOR  (O1 1)  (carry 1)   (a2-out 1)  (a1-out 1))

(Gate  XOR  (X1 2)  (x1-out 2)  (in1 2)     (in2 2))
(Gate  XOR  (X2 2)  (out 2)     (carry 1)   (x1-out 2))
(Gate  AND  (A1 2)  (a1-out 2)  (in1 2)     (in2 2))
(Gate  AND  (A2 2)  (a2-out 2)  (x1-out 2)  (carry 1))
(Gate  IOR  (O1 2)  (carry 2)   (a2-out 2)  (a1-out 2))
 ...
```

Figure 6.3: Scenario Description for the First Two Bits of the Adder

```
;;;; Observations:
(Low (in1 1))      (Low (in2 1))
(Low (in1 2))      (Low (in2 2))
 . . .
(Low (carry 0))    (High (out n))
```

Figure 6.4: Symptomatic Observations for the Adder

```
(Stuck-at-One (X2 <n>))

(Stuck-at-One (X1 <n>))

(Stuck-at-One (O1 <n-1>))

(Stuck-at-One (A2 <n-1>))

(Stuck-at-One (A1 <n-1>))

(Stuck-at-One (O1 <n-2>))
(Stuck-at-One (X1 <n-1>))
 . . .
```

Figure 6.5: First Six Candidates Found for the N-Bit Adder

Figure 6.3 lists the scenario description for the first two bits of the adder. Adjacent bits are linked by the carry output of the lesser bit. Various bit length adders are constructed by asserting the appropriate gate connections, up to the desired last bit.

In the example given in [20], all inputs are low and the output of the nth bit is high. These observations are listed in Figure 6.4. The high output of the nth bit conflicts with the predicted output of 0, based on the assumptions that all components are working correctly. The minimal retraction algorithm finds the same five singleton candidates as in [20], as well as other non-singleton minimal candidates. Figure 6.5 lists the first six candidates found.

Table 6.1 summarizes the performance of the candidate generator for the n-bit adder, for various values of n. Setup time includes constructing all ATMS nodes, assumptions and justifications, but no label propagation. Separate times are given for finding the five singleton candidates and for finding all minimal candidates. A total of $3n - 1$ minimal candidates exist for an n-bit adder; finding all of these becomes prohibitive for n larger than 16. For smaller values of n, the five singleton candidates are found in just a few seconds. PDE finds the five singleton candidates for a 100-bit adder in under 60 seconds. By comparison, in 60 seconds GDE could only diagnose a 4-bit adder, and the original *Sherlock* could diagnose a 6-bit adder in the same time [20]; however, an improved version of *Sherlock* based on the HTMS [20] has diagnosed a 500-bit adder in just six seconds. The run times shown above are roughly $O(n^2)$; this matches

Adder Size			Time (sec)		
#Bits	#Gates	#Candidates	Setup	Singles	All
4	20	11	0.32	2.97	3.92
8	40	23	0.68	4.07	26.23
16	80	47	1.28	6.30	211.72
32	160	95	2.82	13.60	>500
64	320	191	5.75	34.98	>500
128	640	383	11.62	114.18	>500
256	1280	767	23.58	403.23	>500

Table 6.1: Performance Summary for N-Bit Adder Example

```
(==> (and (Gate XOR (X1 ?n)  (x1-out ?n)  (in1 ?n)       (in2 ?n))
          (Gate XOR (X2 ?n)  (out ?n)     (carry ?n-1)   (x1-out ?n))
          (Gate AND (A1 ?n)  (a1-out ?n)  (in1 ?n)       (in2 ?n))
          (Gate AND (A2 ?n)  (a2-out ?n)  (x1-out ?n)    (carry ?n-1))
          (Gate IOR (O1 ?n)  (carry ?n)   (a2-out ?n)    (a1-out ?n)))
     (Assembly (Adder-Bit ?n) (X1 ?n) (X2 ?n) (A1 ?n) (A2 ?n) (O1 ?n)))

(=TR=> (Assembly ?whole . ?parts)
       (iff (OK ?whole) (All-Ok . ?parts)))

(=TR=> (All-Ok ?part)  (OK ?part))
(=TR=> (All-Ok ?part1 ?part2 . ?parts)
       (and (OK ?part1) (All-Ok ?part2 . ?parts)))
```

Figure 6.6: Rules Relating a One-Bit Adder Assembly to Individual Components

the complexity of the "whittling" approach used to find minimal nogoods in CATMS, which for n assumptions must consider the consistency of n environments of average size $n/2$. A better strategy for minimizing conflicts could significantly improve overall performance.

6.2.1 Diagnosing the Adder Bit by Bit

Rather than reasoning at the level of individual logic gates, it is possible to reason about each bit as a module. This decomposition is achieved by making a correctness assumption for each bit, appropriately linked to the correctness of the gates which it comprises. The rules for defining a single bit assembly are given in Figure 6.6. By reasoning at the level of the bit rather than that of the gate, there are fewer combinations of failures to consider.

It is interesting to note that what constitutes a minimal diagnosis depends on the level of reasoning. At the gate-level, the n-bit adder has $3n - 1$ minimal candidates for the low-inputs/high-output scenario; some of these involve a gate failure at every bit position. On the other hand, reasoning at the gate-level yields only two minimal candidates, independent of n:

Adder Size			Time (sec)		
#Bits	#Gates	#Candidates	Setup	Cands	Gates
4	20	2	0.33	0.30	1.87
8	40	2	0.75	0.52	2.18
16	80	2	1.42	1.08	2.87
32	160	2	2.83	2.63	4.90
64	320	2	5.80	7.65	10.98
128	640	2	11.85	24.70	31.47
256	1280	2	23.73	87.95	108.06

Table 6.2: Diagnosing the N-Bit Adder at the Bit Level

the failure must reside in one of the last two bits. All of the candidates at the gate-level involve (at least) one of these two bits. If we could replace these two bit assemblies with good ones, the symptomatic high output would disappear.

Table 6.2 summarizes the performance of the candidate generator for the n-bit adder when reasoning at the bit-level. The two minimal candidates are found in a fraction of the time required at the gate-level to find even the first candidates. Once we have focused on a particular bit, we may open it up by switching to the gate-level. The last column in Table 6.2 indicates the additional time required to identify the five single-gate candidates found within the last two bits. Note that the total time required to find these candidates is significantly less than when reasoning purely at the gate-level.

6.2.2 Discussion

This example demonstrates the performance of the candidate generator in PDE. The run times are respectable, despite the obvious room for improvement in finding minimal nogoods. The 256-bit adder involves literally thousands of assumptions (5122), and yet is diagnosed in only a few minutes. This level of performance would be impossible without the ability in CATMS to focus on relevant contexts.

This example also illustrates the benefits of hierarchical reasoning in diagnosis. By reasoning about the adder at the bit-level rather than the gate-level, run-times are significantly reduced, along with the number of minimal candidates.

The rules used for the adder example do not constitute a causal model; they simply enforce the logical constraints observed by each type of gate. Because the model is not causal, there are no closed-world assumptions to violate, and no abduction required to identify missing structure. The remaining examples all utilize a causal domain theory, so that closed-world assumptions and the GA3 abduction component of PDE come into play.

6.3 Two Light Bulbs and a Battery

The simple circuit shown in Figure 6.7 has appeared in various forms throughout the model-based diagnosis literature [75, 27, 44]; it has been used to illustrate the need for explicit fault

Scenario Description:

```
(Battery BATT1)
(Light_Bulb BULB1)
(Light_Bulb BULB2)
(Connection BATT1 TERM1 TERM2)
(Connection BULB1 TERM1 TERM2)
(Connection BULB2 TERM1 TERM2)
```

Observations:

```
(not (Glowing BULB1))
(Glowing BULB2)
```

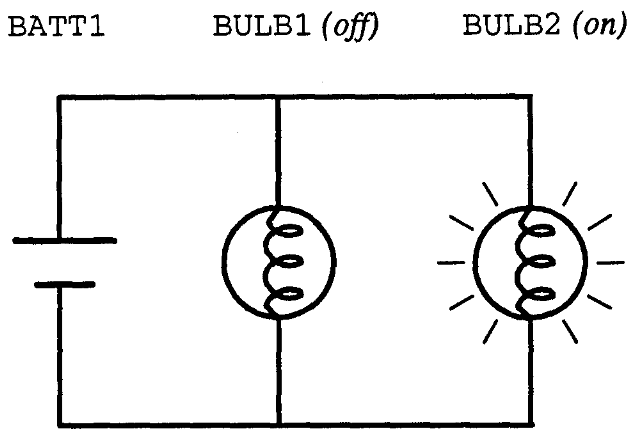


Figure 6.7: Two Light Bulbs and a Battery in Parallel

models to rule out counterintuitive candidates. Here we demonstrate how the assumption of a complete domain theory can perform the same role as the explicit fault models of [75], or the physical impossibility axioms of [44]. The circuit shown in Figure 6.7 consists of two light bulbs and a battery connected in parallel. One of the bulbs is observed to be glowing while the other is not glowing. Without an explicit fault model for light bulbs, most diagnostic systems find two minimal candidates: either the unlit bulb is bad,[2] or the battery is dead and the lit bulb is "stuck on". This second explanation is quickly dismissed as counterintuitive by a human troubleshooter, but the consistency-based approach has no basis for dismissing it. Lacking explicit failure modes, the typical model for this example does not preclude light bulbs from failing in such a way that they glow with no voltage applied, and it provides no insights into what might be causing the one bulb to glow, assuming the battery is dead.

Some researchers [75, 44] have succeeded in eliminating the second candidate above, by adding the constraint the a broken (or disconnected) bulb does not glow. We argue that the goal should not be to eliminate the candidate, but to explain it. After all, it is possible to modify the circuit (e.g., by cutting a wire and connecting a second battery) such that the candidate is correct. Rather than eliminate the candidate completely, we can dismiss it based on the unlikely nature of its possible causes.

Figure 6.8 lists a simple domain theory for light bulbs. Batteries and light bulbs are modeled as components having current and voltage quantities. Components are connected to pairs of terminals, which can have a voltage across them. A light bulb is a type of resistor, which defines a relationship between its current and voltage (Ohm's Law). Two process types: `Current_Flow` and `Glowing`, influence the amount of heat in a component. In this admittedly limited theory, light bulbs are the only objects capable of glowing, and only when the bulb has a nonzero current flow.

[2]This includes the possibility that the bulb is disconnected. Some models—including the one described here—distinguish between a bad bulb and a bad connection.

```
(defPredicate (Connection ?comp ?term1 ?term2)
  (:= (voltage ?comp) (- (voltage ?term1) (voltage ?term2))))

(defEntity Battery      ;; A battery has positive voltage:
  (Component ?self)  (Positive (voltage ?self))  (Has_Potential ?self))

(defEntity Light_Bulb  ;; A light bulb is a special kind of resistor:
  (Resistor ?self))

(defEntity Resistor     ;; Has positive resistance and obeys Ohm's Law:
  (Component ?self)  (not (Battery ?self))
  (Positive (resistance ?self))
  (:= (current ?self) (/ (voltage ?self) (resistance ?self))))

(defEntity Component    ;; Components have voltage, current and power:
  (when (not (Has_Potential ?self))  (Zero (voltage ?self)))
  (:= (power ?self) (* (voltage ?self) (current ?self))))

(defProcess (Current_Flow ?comp)  ;; During current flow, power influences heat:
  Individuals ((?comp :type Component))
  Conditions  ((Non-Zero (current ?comp)))
  Influences  ((I+ (heat ?comp) (power ?comp))))

(defProcess (Glowing ?bulb)        ;; Bulbs glow during current flow:
  Individuals ((?bulb :type Light_Bulb))
  Instance_of ((Current_Flow ?bulb))
  Influences  ((I- (heat ?bulb) (power ?bulb))))

(defClosed-Predicate Has_Potential)          ;; No potential by default;

(==> (and (Connection ?comp ?term1 ?term2)  ;; Connected components inherit potential;
          (Has_Potential ?term1 ?term2))
     (Has_Potential ?comp))

(==> (and (Connection ?comp ?term1 ?term2)  ;; Potential propagates along connections:
          (Has_Potential ?term2 ?term3)
          (:TEST (not (equal ?term1 ?term3))))
     (Has_Potential ?term1 ?term3))

(==> (and (Connection ?batt ?term1 ?term2)  ;; A battery has potential:
          (Battery ?batt))
     (Has_Potential ?term1 ?term2))
```

Figure 6.8: A Simple Domain Theory for Light Bulbs

```
Environment <E-71*>:
  (Battery BATT1)
  (Light_Bulb BULB1)
  (Light_Bulb BULB2)
  (Connection BATT1 TERM1 TERM2)
  (Connection BULB1 TERM1 TERM2)
  (Connection BULB2 TERM1 TERM2)
  (CWA 1:  (Affected (heat BULB1) -))
  (CWA 2:  (Affected (heat BULB1) +))
  (CWA 3:  (Affected (heat BULB2) -))
  (CWA 4:  (Affected (heat BULB2) +))
  (CWA 5:  (Has_Potential TERM1 TERM2))
  (CWA 6:  (Has_Potential BULB2))
  (CWA 7:  (Has_Potential BULB1))
  (CWA 8:  (Has_Potential BATT1))
  (not (Glowing BULB1))
  (Glowing BULB2)
```

Figure 6.9: Base Environment for the Light Bulb Example

A component which is directly or indirectly connected to a battery has potential, indicating that it is possible for it to have a nonzero voltage. The `Has_Potential` predicate is closed to enforce the constraint that by default components do not have potential. This is the key in this domain theory to preventing an isolated light bulb from having a positive voltage, and thus glow all by itself.[3]

6.3.1 Candidate Generation

Figure 6.9 shows the base environment consisting of the scenario description, the observations and the closed-world assumptions made while composing the model. This environment is contradictory, indicating a conflict between predicted and observed behavior; specifically, the model predicts that BULB1 should be glowing but it is not. This environment is given to the minimal retraction algorithm, which generates the four candidates shown in Figure 6.10. These are the only minimal candidates for this example. The first two candidates correspond to the two most intuitive explanations: either the bulb has become disconnected, or it is no longer a functioning light bulb (i.e., it is "blown").

The last two candidates blame BULB1's failure to glow on the battery, suggesting that it is either disconnected or inoperative (i.e., "dead"). The fact that BULB2 is glowing requires it to have a potential; this in turn requires that it is disconnected from the rest of the circuit (which by assumption has no potential), and that "something" provides it with potential. This unknown "something" is represented by the violated closed-world assumption for `(Has_Potential BULB2)`.

[3]A more fine-grained domain theory might enforce this by modeling current as charge flow, and enforcing Kirchoff's Current Law of no charge accumulation at a terminal. Such a change to the model would not significantly alter the candidates generated—only how they are found.

```
<E-217> - <E-71*>:
  (not (Connection BULB1 TERM1 TERM2))

<E-224> - <E-71*>:
  (not (Light_Bulb BULB1))

<E-265> - <E-71*>:
  (not (Connection BATT1 TERM1 TERM2))
  (not (Connection BULB2 TERM1 TERM2))
  (not (CWA 6:  (Has_Potential BULB2)))

<E-243> - <E-71*>:
  (not (Battery BATT1))
  (not (Connection BULB2 TERM1 TERM2))
  (not (CWA 6:  (Has_Potential BULB2)))
```

Figure 6.10: Four Candidates for the Light Bulb Example

6.3.2 Abductive Backchaining

The last two candidates above both contain a violated closed-world assumption, indicating that some unknown structure is providing potential to BULB2. We use abductive backchaining to identify this unknown structure. There are an infinite number of circuits capable of providing potential to BULB2, so we cannot enumerate them all. The GA3 abduction algorithm is capable of generating these circuits, one at a time and in best-first order.

Figure 6.11 shows the justifications constructed for the goal: (Has_Potential BULB2). This example demonstrates that the reuse of existing justifications for variants of an abductive goal is essential for domains such as this one, where recursive rules would otherwise lead to an infinite tree of justifications. By linking into existing variant goals, a finite justification structure acts as a generator for an infinite set of explanations.

Figure 6.12 lists the first three environments[4] propagated to the top-level goal. The first explanation suggests that there exists a battery (SK-BATT-3) which is connected to BULB2 through a pair of terminals (SK-TERM-1 and SK-TERM-2). Subsequent explanations are similar, but involve an indirect connection through one or more intervening components. Because these explanations are generated best-first, there is no need to generate them all (even if we could!); the first one is already sufficiently unlikely to warrant the dismissal of this candidate.

6.3.3 Discussion

This example illustrates the difference between process-based diagnosis and the consistency-based approach common in model-based diagnosis. The defaulting of all domain-theory constructs to false means that only the indicated objects under the specified circumstances are

[4]Environments generated during abduction are pretty-printed with all binding assumptions applied and removed.

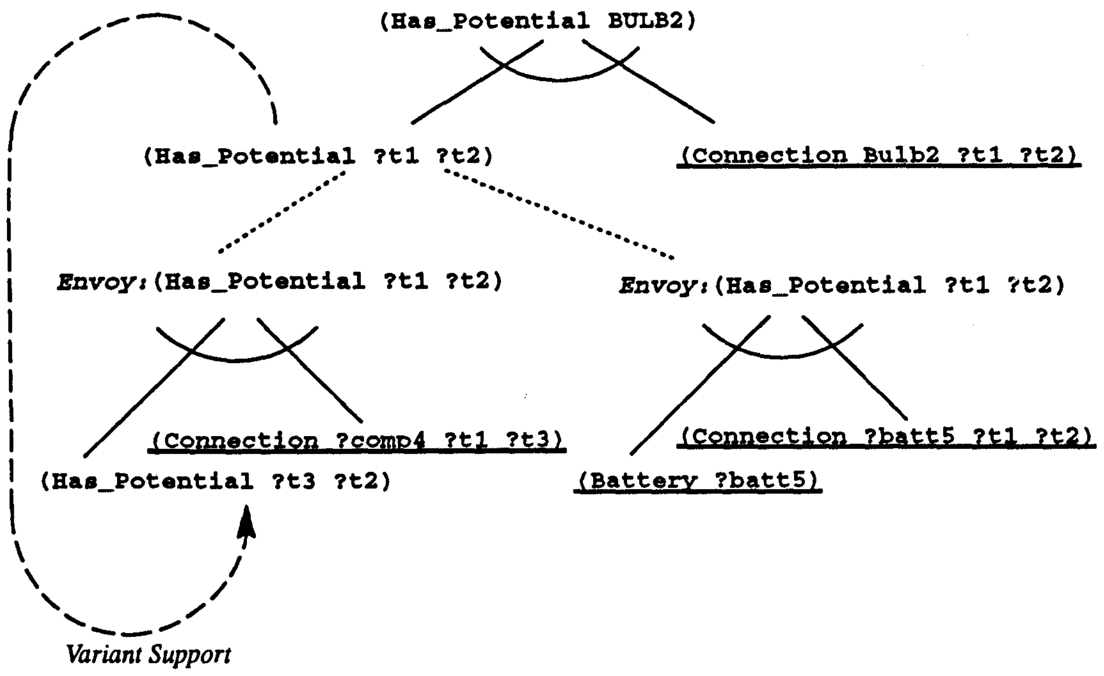

(Has_Potential BULB2)
(Has_Potential ?t1 ?t2)
(Connection Bulb2 ?t1 ?t2)
Envoy: (Has_Potential ?t1 ?t2)
Envoy: (Has_Potential ?t1 ?t2)
(Connection ?comp4 ?t1 ?t3)
(Has_Potential ?t3 ?t2)
(Connection ?batt5 ?t1 ?t2)
(Battery ?batt5)
Variant Support

Figure 6.11: Justifications for Abduction on (Has_Potential BULB2)

```
Environment <E-363>:
  (Connection BULB2 SK-TERM-2 SK-TERM-1)
  (Connection SK-BATT-3 SK-TERM-2 SK-TERM-1)
  (Battery SK-BATT-3)

Environment <E-517>:
  (Connection BULB2 SK-TERM-2 SK-TERM-1)
  (Connection SK-COMP-5 SK-TERM-2 SK-TERM-4)
  (Battery SK-BATT-6)
  (Connection SK-BATT-6 SK-TERM-4 SK-TERM-1)

Environment <E-621>:
  (Connection BULB2 SK-TERM-2 SK-TERM-1)
  (Connection SK-COMP-5 SK-TERM-2 SK-TERM-4)
  (Connection SK-BATT-9 SK-TERM-8 SK-TERM-1)
  (Battery SK-BATT-9)
  (Connection SK-COMP-7 SK-TERM-4 SK-TERM-8)
```

Figure 6.12: First Three Explanations Found for (Has_Potential BULB2)

Solution Phase	#Nodes	#Asns	#Envs	#Clauses	#Solutions	Time (sec)
Model Instantiation	298	36	73	311	—	2.98
Cand. Generation	0	4	202	3	4	7.48
Abduction Setup	10	4	5	7	—	0.28
Abduction	240	22	121	225	1st	3.92
Total	548	66	401	546	4	14.66

Table 6.3: Performance Summary for the Light Bulb Example

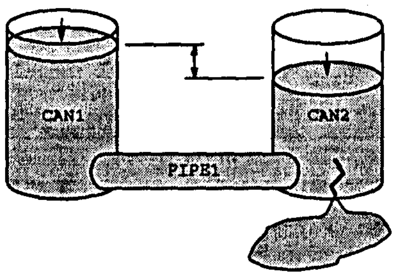
capable of a given behavior (such as glowing). In previous models, voltage was a sufficient condition for a bulb to glow (assuming a functioning light bulb), but not a necessary condition for glowing in general. In this closed domain theory, a nonzero voltage is both a necessary and sufficient condition for a bulb to glow (assuming the bulb is intact). By removing the voltage from the light bulbs, we remove their reason to glow. And, if BULB2 is no longer a light bulb, then it cannot be glowing, as nothing except a light bulb (in this domain theory) can glow.

The model-closing mechanism required for compositional, process-centered models provides a much more natural way to express these constraints than either explicit fault models [75] or physical impossibility axioms [75]. Rather than having to enumerate everything which is not possible, we specify the possible and make the assumption that we have been completely thorough in doing so. If it were possible for a light bulb or other object to glow in some other way then we would have said so in the domain theory.

Table 6.3 summarizes the run-times and ATMS statistics for each phase of diagnosis for the light bulb example. Due to the simplicity of this domain, the diagnosis proceeds very quickly. Subsequent examples involve the more complicated domain of thermodynamics, as their respective statistics reflect.

Scenario Description:

```
(Container CAN1) (Pipe PIPE1) (Container CAN2)
(Connection PIPE1 (bottom CAN1) (bottom CAN2))
```



Observations:

```
(Positive (volume (liquid_in CAN1)))
(Positive (volume (liquid_in CAN2)))
(> (level (liquid_in CAN1))
   (level (liquid_in CAN2)))
(Decreasing (level (liquid_in CAN1)))
(Decreasing (level (liquid_in CAN2)))
```

Figure 6.13: A Leak in One of Two Containers

6.4 A Leaking Container

This is the first of several examples drawn from the fluids domain; these examples utilize the domain theory listed in Appendix A. This simple example illustrates how a failure can cause new entities and new process instances to come into existence. Figure 6.13 depicts a scenario consisting of two containers of water connected by a pipe. This scenario is familiar to anyone acquainted with QP theory; it is the simplest scenario within the fluids domain which includes active processes. The observations suggest that something is wrong; the level of liquid

```
(defModel_Asn (Consider Gravity)                       :TRUE)
(defModel_Asn (Consider (C_S * * *))                   :TRUE)
(defModel_Asn (Consider (Substance WATER))             :TRUE)
(defModel_Asn (Consider (State LIQUID))                :TRUE)
(defModel_Asn (Consider (State GAS))                   :TRUE)
(defModel_Asn (Disallow (Zero_Mass OB*))               :TRUE)
(defModel_Asn (Distinguish (Can_Status CAN*))          :FALSE)
(defModel_Asn (Consider (Fluid_Conductance PIPE*)) :FALSE)
(defModel_Asn (Consider (Boiling_In CAN*))             :TRUE)
(defModel_Asn (Consider (Condensing_In CAN*))          :TRUE)
(defModel_Asn (Consider Complex_boiling)               :FALSE)
(defModel_Asn (Consider Variable_Boiling_Point)        :FALSE)
(defModel_Asn (Consider (Level_Path PIPE*))            :TRUE)
(defModel_Asn (Consider (Aligned PIPE*))               :TRUE)
(defModel_Asn (Consider (On PUMP*))                    :TRUE)
```

Figure 6.14: Default Modeling Assumptions for Fluids Examples

in CAN2 is dropping, despite the lack of any known process which could be influencing its level downward. The model fails to predict the falling level of liquid in CAN2, and instead predicts a rising level, due to the flow through PIPE1.

6.4.1 Candidate Generation

Diagnosing this example begins by adding the observations given in Figure 6.13 into the set of base assumptions shown in Figure 6.15. The base assumptions consist of the scenario description and the modeling assumptions, together with the closed-world assumptions made while composing the model. These base assumptions define the default beliefs for this and the next two examples; candidates for these examples consist of minimal retractions from this set.[5] Figure 6.16 shows the environment resulting from adding the observations to the set of base assumptions; for conciseness, only the differences from the base environment are shown. The asterisk indicates that the environment is contradictory—that is, that the observations are inconsistent with the predictions of the model.

The minimal retraction algorithm of Chapter 4 finds a single candidate, shown in Figure 6.17. The candidate indicates a violated closed-world assumption for the set of negative direct influences on the mass of the contained liquid in CAN2. The amount of liquid must be influenced negatively, contrary to the original model.

6.4.2 *Abductive Backchaining*

The violated closed-world assumption in this example suggests that some unknown structure may have instantiated a process which is influencing the level of liquid downward. The fact that no other candidates exist indicate that this is the only possibility. The violated closed-world assumption in itself does not constitute an acceptable diagnosis, because it does not identify the underlying physical structure of the device after failure. We want to know what is responsible for the negative influence, not merely that it exists. Abductive backchaining on the rules of the domain theory yields the missing structural entities and connections required to explain the negative influence.

The GA3 abduction algorithm described in Chapter 5 is given the top-level goal:
(Affected (net_influence (mass (C-S WATER LIQUID CAN2))) -).
Backchaining on the rules of the domain theory, GA3 constructs a tree of justifications whose root is the top-level goal. Figure 6.18 shows the top portion of the resulting justification tree. Some process must negatively influence the amount of liquid in CAN2. The only known process types capable of influencing the mass of a contained liquid are fluid flows and phase transitions.

Once the justifications have been built, GA3 begins propagating label environments best-first toward the top-level goal. This propagation eventually yields the four explanations given in Figure 6.19. The first explanation suggests that some hot object (SK-SRC-17) may be in thermal contact with the liquid, causing it to boil away. The second explanation has a pump connected to CAN2, draining its contents.

The last two explanations are most naturally interpreted as a leak. An active fluid flow process transfers liquid from CAN2 to an unknown destination (e.g., the floor), through an unknown path (e.g., a crack in the container). The two environments differ in the directed connectivity of the fluid path. The exact location of the portal as well as the identity of the

[5] Again, we ignore candidates in which the observations are retracted.

```
Environment <E-95>:
  (Container CAN1)  (Container CAN2)
  (Pipe P1)  (Aligned P1)
  (Connection P1 (bottom CAN1) (bottom CAN2))
  (= (height (bottom CAN2)) (height (bottom CAN1)))
  (CWA 1:  (Influenced (temperature (C_S WATER LIQUID CAN1) ABS) -))
  (CWA 2:  (Influenced (temperature (C_S WATER LIQUID CAN1) ABS) +))
  (CWA 3:  (Influenced (temperature (boil (C_S WATER LIQUID CAN1)) ABS) -))
  (CWA 4:  (Influenced (temperature (boil (C_S WATER LIQUID CAN1)) ABS) +))
  (CWA 5:  (Influenced (volume (liquid_in CAN1)) -))
  (CWA 6:  (Influenced (volume (liquid_in CAN1)) +))
  (CWA 7:  (Influenced (heat (C_S WATER LIQUID CAN1)) -))
  (CWA 8:  (Influenced (heat (C_S WATER LIQUID CAN1)) +))
  (CWA 9:  (Influenced (pressure (gas_in CAN1) ABS) -))
  (CWA 10: (Influenced (pressure (gas_in CAN1) ABS) +))
  (CWA 11: (Influenced (temperature (C_S WATER LIQUID CAN2) ABS) -))
  (CWA 12: (Influenced (temperature (C_S WATER LIQUID CAN2) ABS) +))
  (CWA 13: (Influenced (temperature (boil (C_S WATER LIQUID CAN2)) ABS) -))
  (CWA 14: (Influenced (temperature (boil (C_S WATER LIQUID CAN2)) ABS) +))
  (CWA 15: (Influenced (volume (liquid_in CAN2)) -))
  (CWA 16: (Influenced (volume (liquid_in CAN2)) +))
  (CWA 17: (Influenced (heat (C_S WATER LIQUID CAN2)) -))
  (CWA 18: (Influenced (heat (C_S WATER LIQUID CAN2)) +))
  (CWA 19: (Influenced (pressure (gas_in CAN2) ABS) -))
  (CWA 20: (Influenced (pressure (gas_in CAN2) ABS) +))
  (CWA 21: (Influenced (net_influence (mass (C_S WATER LIQUID CAN2))) -))
  (CWA 22: (Influenced (net_influence (mass (C_S WATER LIQUID CAN2))) +))
  (CWA 23: (Influenced (net_influence (mass (C_S WATER LIQUID CAN1))) -))
  (CWA 24: (Influenced (net_influence (mass (C_S WATER LIQUID CAN1))) +))
  (CWA 25: (Wet (bottom CAN2)))
  (CWA 26: (Wet (bottom CAN1)))
  (CWA 27: (Affected (net_influence (mass (C_S WATER LIQUID CAN1))) -))
  (CWA 28: (Affected (net_influence (mass (C_S WATER LIQUID CAN1))) +))
  (CWA 29: (Affected (net_influence (mass (C_S WATER LIQUID CAN2))) -))
  (CWA 30: (Affected (net_influence (mass (C_S WATER LIQUID CAN2))) +))
  (CWA 31: (:= (temperature (C_S WATER LIQUID CAN2) ABS)
              (/ (heat (C_S WATER LIQUID CAN2)) (mass (C_S WATER LIQUID CAN2)))))
  (CWA 32: (:= (temperature (C_S WATER LIQUID CAN1) ABS)
              (/ (heat (C_S WATER LIQUID CAN1)) (mass (C_S WATER LIQUID CAN1)))))
```

Figure 6.15: Base Environment for the Two-Container Scenario

```
Environment <E-97*> - <E-95>:
  (Positive (mass (C_S WATER LIQUID CAN1)))
  (Positive (mass (C_S WATER LIQUID CAN2)))
  (Positive (volume (liquid_in CAN1)))
  (Positive (volume (liquid_in CAN2)))
  (Decreasing (level (liquid_in CAN1)))
  (Decreasing (level (liquid_in CAN2)))
  (> (level (liquid_in CAN1))  (level (liquid_in CAN2)))
  (< (height (bottom CAN1))  (level (liquid_in CAN1)))
  (< (height (bottom CAN2))  (level (liquid_in CAN2)))
```

Figure 6.16: Contradictory Base Environment for the Leak Example

```
<Env-188> - <Env-97*>:
  (not (CWA 29: (Affected (net_influence (mass (C_S WATER LIQUID CAN2))) -)))
```

Figure 6.17: A Single Candidate for the Leak Example

path and the destination of the flow remain unspecified. However, the environments do indicate that the portal must be somewhere below the liquid's surface, and the destination must have a lower pressure than that of CAN2.

A measurement of the liquid's temperature can rule out the possibility of boiling. The environment corresponding to boiling in Figure 6.19 implies that the temperature is at the boiling point, and that some heat source is being applied. If the temperature is measured to be below the boiling point, then the boiling hypothesis is rejected. In fact, by including the observation that the liquid is too cold to boil, the boiling hypothesis is never given serious consideration, and the abductive search is more focused.[6]

6.4.3 Discussion

This example demonstrates the two-tiered approach which characterizes process-based diagnosis. In the first step, the traditional consistency-based approach is used to identify minimal departures from the original model which are consistent with observations. In this example only one minimal candidate is found, representing the fact that something must be causing the liquid level to drop. In the second step, abductive backchaining is used to explain the violated closed-world assumption from the first step. This unique combination of diagnostic approaches yields greater efficiency and reasoning power than either approach alone.

[6]Including the low-temperature observation reduces the run-time for abductive search by twelve seconds, or about ten percent.

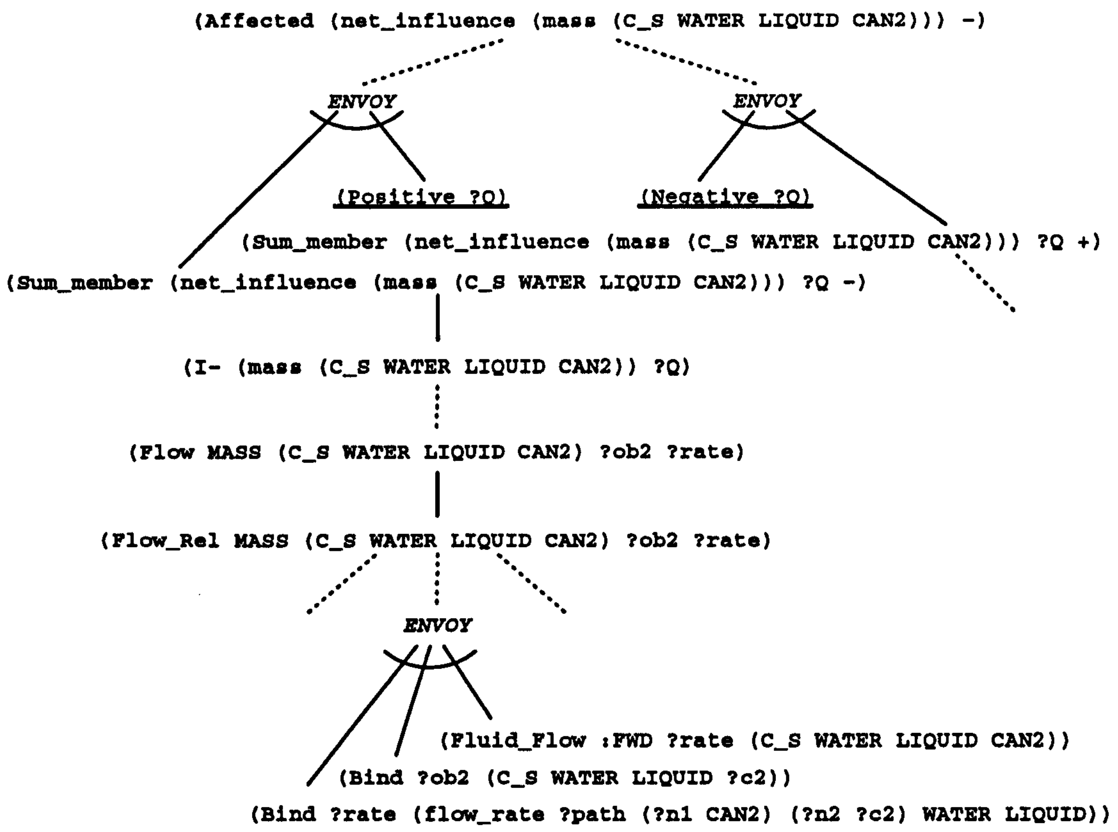

(Affected (net_influence (mass (C_S WATER LIQUID CAN2))) -)
ENVOY
ENVOY
(Positive ?Q)
(Negative ?Q)
(Sum_member (net_influence (mass (C_S WATER LIQUID CAN2))) ?Q +)
(Sum_member (net_influence (mass (C_S WATER LIQUID CAN2))) ?Q -)
(I- (mass (C_S WATER LIQUID CAN2)) ?Q)
(Flow MASS (C_S WATER LIQUID CAN2) ?ob2 ?rate)
(Flow_Rel MASS (C_S WATER LIQUID CAN2) ?ob2 ?rate)
ENVOY
(Fluid_Flow :FWD ?rate (C_S WATER LIQUID CAN2))
(Bind ?ob2 (C_S WATER LIQUID ?c2))
(Bind ?rate (flow_rate ?path (?n1 CAN2) (?n2 ?c2) WATER LIQUID))

Figure 6.18: Abductive Justifications for a Falling Level in CAN2.

The traditional consistency-based approach is too weak in that it finds too many candidates. All of the components in the scenario contribute to the prediction that the level in CAN2 should be rising, so the consistency-based approach would find a candidate for each component. The assumption that the domain theory is complete allows us to take a stronger stand, and negate component assumptions rather than merely retracting them. Any new structure created by the failure is detected as violated closed-world assumptions, which are then explained abductively.

A pure abductive approach would be both less efficient and less powerful. This is the motivation for explaining violated closed-world assumptions, rather than the observations themselves. Had we given the observations to GA3 as the top-level goal, it would have failed to find any explanations. This is due to the inherent ambiguity of the qualitative model; abductive backchaining on the current set of domain-independent constraints is unable to prove that a quantity should be decreasing, even though it is influenced positively. Reasoning about competing influences can only be done procedurally, after the complete sets of positive and negative influences are known. By backchaining on an intermediate goal between behavior and structure, abduction is both more efficient and more likely to find an explanation.

In addition to providing the goal for abduction, the candidate generator provides a consistent context in which to focus the abductive search. The original set of beliefs shown in

```
Environment <E-367>:
  (Container CAN2)
  (Physob SK-SRC-17)
  (Heat_Connection SK-HPATH-16 SK-SRC-17 (C_S WATER LIQUID CAN2))
  (Positive (Generation_Rate GAS CAN2))
  (= (temperature (C_S WATER LIQUID CAN2) ABS)
     (temperature (boil (C_S WATER LIQUID CAN2)) ABS))
  (< (temperature (C_S WATER LIQUID CAN2) ABS)
     (temperature SK-SRC-17 ABS))
  (Positive (heat SK-SRC-17))

Environment <E-370>:
  (Container CAN2)
  (Fluid_Pump SK-PATH-14)
  (Connection SK-PATH-14 (SK-N-13 CAN2) (SK-N-12 SK-C-11))
  (On SK-PATH-14)
  (Portal (SK-N-13 CAN2))
  (Positive (flow_rate SK-PATH-14 (SK-N-13 CAN2) (SK-N-12 SK-C-11) WATER LIQUID))
  (<  (height (SK-N-13 CAN2)) (height (liquid_in CAN2)))

Environment <E-869>:
  (Container CAN2)
  (Pipe SK-PATH-14)
  (Connection SK-PATH-14 (SK-N-13 CAN2) (SK-N-12 SK-C-11))
  (Portal (SK-N-13 CAN2))
  (not (Consider (fluid_conductance SK-PATH-14)))
  (Aligned SK-PATH-14)
  (<  (pressure (SK-N-12 SK-C-11) ABS) (pressure (SK-N-13 CAN2) ABS))
  (Positive (flow_rate SK-PATH-14 (SK-N-13 CAN2) (SK-N-12 SK-C-11) WATER LIQUID))
  (<  (height (SK-N-13 CAN2)) (height (liquid_in CAN2))))

Environment <E-872>:
  (Container CAN2)
  (Pipe SK-PATH-24)
  (Connection SK-PATH-24 (SK-N-23 SK-C-22) (SK-N-21 CAN2))
  (Portal (SK-N-21 CAN2))
  (not (Consider (fluid_conductance SK-PATH-24)))
  (Aligned SK-PATH-24)
  (< (pressure (SK-N-23 SK-C-22) ABS) (pressure (SK-N-21 CAN2) ABS))
  (Negative (flow_rate SK-PATH-24 (SK-N-23 SK-C-22) (SK-N-21 CAN2) WATER LIQUID))
  (< (height (SK-N-21 CAN2)) (height (liquid_in CAN2)))
```

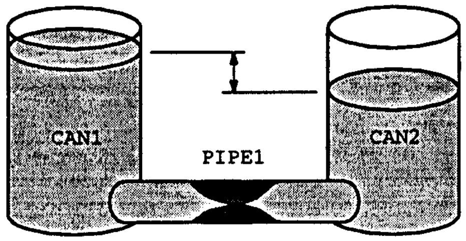
Figure 6.19: Four Environments Found by Abductive Search

Solution Phase	#Nodes	#Asns	#Envs	#Clauses	#Solutions	Time (sec)
Model Instantiation	1158	76	97	1191	—	8.85
Cand. Generation	0	0	91	4	1*	4.52
Abduction Setup	1034	86	108	1080	—	12.67
Abduction (first)	2825	93	1245	3085	2	14.42
Abduction (rest)	1249	48	948	1388	2	88.43
Total	6266	324	2489	6748	4	128.89

Table 6.4: Performance Summary for the Leak Example

Scenario Description:

```
(Container CAN1)  (Pipe PIPE1)  (Container CAN2)
(Connection PIPE1 (bottom CAN1) (bottom CAN2))
```



Observations:

```
(Positive (volume (liquid_in CAN1)))
(Positive (volume (liquid_in CAN2)))
(> (level (liquid_in CAN1))
   (level (liquid_in CAN2)))
(Constant (level (liquid_in CAN1)))
(Constant (level (liquid_in CAN2)))
```

Figure 6.20: A Clogged Pipe Connecting Two Containers

Figure 6.16 is inconsistent, and so cannot be used as a focusing context.[7] The ability to focus on a particular class of explanation helps to further minimize the potentially-high cost of abductive backchaining.

Table 6.4 summarizes performance on the leak example. Each row of the table describes a separate phase of problem solving. Because the only consistent candidate involves a closed-world assumption violation, no true diagnoses are available until completion of the first round of abduction—about 40 seconds into the problem. Finding the two remaining explanations requires an additional 88 seconds.

6.5 A Clogged Pipe

Figure 6.20 depicts a scenario consisting of two containers connected by a pipe. This example is interesting because it shows how a failure can cause part of the scenario description and a

[7]Anything is deducible given a contradiction.

```
<E-97*> - <E-95>:
  (Positive (mass (C_S WATER LIQUID CAN1)))
  (Positive (mass (C_S WATER LIQUID CAN2)))
  (Positive (volume (liquid_in CAN1)))
  (Positive (volume (liquid_in CAN2)))
  (Constant (level (liquid_in CAN1)))
  (Constant (level (liquid_in CAN2)))
  (> (level (liquid_in CAN1)) (level (liquid_in CAN2)))
  (< (height (bottom CAN1)) (level (liquid_in CAN1)))
  (< (height (bottom CAN2)) (level (liquid_in CAN2)))
```

Figure 6.21: Base Environment for the Clogged Pipe Example

corresponding part of the resulting process structure to disappear from the model. An imbalance in the levels of liquid in the two containers normally results in a flow of liquid from the higher to the lower level. In this example, a clog in the pipe blocks the flow.

The levels of liquid in the two containers is observed to be constant, contrary to the predictions of the model. Figure 6.21 shows the contradictory base environment for this example. In addition to the observations shown, the base environment contains all of the scenario description, modeling assumptions and closed-world assumptions for the instantiated model, shown previously in Figure 6.15. Based on the observation that the liquid levels are unequal, the model predicts that a fluid flow from the higher to the lower level will cause the higher level to drop and the lower level to rise. The set of base assumptions is inconsistent because a level cannot be both constant and changing at the same time.

6.5.1 Candidate Generation

Given the inconsistent set of base assumptions from Figure 6.21, the candidate generator finds three singleton candidates—all three involving PIPE1. Either the pipe is missing, or it is present but disconnected, or it is not aligned. These candidates all represent a nullifying failure of the pipe (described in Section 3.6), whose most natural interpretation is that the pipe is clogged. These first three candidates constitute valid structural explanations of the failure, as they do not involve any closed-world assumption violations. The original scenario description minus the pipe or its connection provide acceptable explanations, whose composed models predict the observed behavior (i.e., constant liquid levels). Note that the presence of the modeling assumption for pipe alignment provides an alternative explanation of the failure. This modeling assumption could be viewed either as representing a failure mode for pipes, or simply as a shorthand way to reason about valves.

Should the first three candidates be rejected for some reason, then the generator finds a fourth candidate, consisting of two closed-world assumption violations. This candidate represents the possibility that some other process (or combination of processes) exactly counteracts the influences of the liquid flow. This is a legitimate possibility; for example, there could be a pump returning the liquid to the upstream container. The candidate generator indicates that no other possibilities exist. The four candidates found are shown in Figure 6.22.

```
<E-209> - <E-97*>:
  (not (Pipe PIPE1))

<E-189> - <E-97*>:
  (not (Aligned PIPE1))

<E-168> - <E-97*>:
  (not (Connection PIPE1 (bottom CAN1) (bottom CAN2)))

<E-245> - <E-97*>:
  (not (CWA 28: (Affected (net_influence (mass (C_S WATER LIQUID CAN1))) +)))
  (not (CWA 29: (Affected (net_influence (mass (C_S WATER LIQUID CAN2))) -)))
```

Figure 6.22: Four Candidates for the Clogged Pipe Example

```
(and (Affected (net_influence (mass (C_S WATER LIQUID CAN1))) +)
     (Affected (net_influence (mass (C_S WATER LIQUID CAN2))) -))
```

Figure 6.23: Goal for Abductive Search for the Clogged Pipe Example

6.5.2 Abductive Backchaining

The last candidate found above does not describe physical structure and so requires completion using abductive search. Even before its completion, however, the candidate makes certain behavioral predictions which provide a basis for its dismissal. For example, if the pipe is still present and connected, then there must be a nonzero flow through it from the high to the low level. A probe on the flow rate through the pipe could discriminate among these candidates. In addition, some other process(es) must be positively influencing the high level and negatively influencing the low level.

In order to further explain the fourth candidate, the abduction algorithm is given the conjunctive goal of deriving a positive influence on CAN1 and a negative influence on CAN2. The exact goal is shown in Figure 6.23. Figure 6.24 shows the first two environments found by abductive search. The first environment may be interpreted as two separate pumps: one pumping water into CAN1 and the other pumping water out of CAN2. Alternatively, the two Skolem pumps may be unified into one pump, which is moving water from CAN2 to CAN1.[8] The second environment suggests that a pump is providing water to CAN1; the water then flows into CAN2 where it is boiled away. The rates of the three processes exactly match, so that the amount of water in each container is constant. GA3 eventually finds a total of sixteen explanations, representing the cross-product of the four explanations found independently for each of the two goals (as shown in Figure 6.19).

[8]This unification step is not currently performed, but could be automated, as described in [57].

```
Environment <E-578>:
  (Container SK-C-20)
  (Container CAN2)
  (Fluid_Pump SK-PATH-14)
  (Fluid_Pump SK-PATH-23)
  (Connection SK-PATH-14 (SK-N-13 CAN2)    (SK-N-12 SK-C-11))
  (Connection SK-PATH-23 (SK-N-21 SK-C-20) (SK-N-22 CAN1))
  (On SK-PATH-14)
  (On SK-PATH-23)
  (Portal (SK-N-13 CAN2))
  (Portal (SK-N-21 SK-C-20))
  (<  (height (SK-N-13 CAN2))    (height (liquid_in CAN2)))
  (<  (height (SK-N-21 SK-C-20)) (height (liquid_in SK-C-20)))
  (Positive (flow_rate SK-PATH-14 (SK-N-13 CAN2) (SK-N-12 SK-C-11) WATER LIQUID))
  (Positive (flow_rate SK-PATH-23 (SK-N-21 SK-C-20) (SK-N-22 CAN1) WATER LIQUID))

Environment <E-642>:
  (Container SK-C-20)
  (Container CAN2)
  (Physob SK-SRC-17)
  (Fluid_Pump SK-PATH-23)
  (Connection SK-PATH-23 (SK-N-21 SK-C-20) (SK-N-22 CAN1))
  (On SK-PATH-23)
  (Portal (SK-N-21 SK-C-20))
  (Heat_Connection SK-HPATH-16 SK-SRC-17 (C_S WATER LIQUID CAN2))
  (>= (temperature (C_S WATER LIQUID CAN2) ABS)
      (temperature (boil (C_S WATER LIQUID CAN2)) ABS))
  (<  (temperature (C_S WATER LIQUID CAN2) ABS)
      (temperature SK-SRC-17 ABS))
  (<  (height (SK-N-21 SK-C-20)) (height (liquid_in SK-C-20)))
  (Positive (heat SK-SRC-17))
  (Positive (Generation_Rate GAS CAN2))
  (Positive (flow_rate SK-PATH-23 (SK-N-21 SK-C-20) (SK-N-22 CAN1) WATER LIQUID))
```

Figure 6.24: The First Two Environments Found by Abductive Search

6.5.3 Discussion

The three singleton candidates found for this example all point to a failure in the fluid path connecting the two containers. Either the pipe has failed outright, or it is disconnected, or it is not aligned. In one way or another, the pipe is failing to provide a fluid connection between the two containers. As far as the behavior of the liquid is concerned, the pipe is effectively absent. The default belief that the pipe is aligned is a kind of modeling assumption, so this example demonstrates how a change in a modeling assumption can correct a failed prediction.

Real pipes in real fluid systems do not typically become disconnected from containers without introducing other paths at the same time; this kind of knowledge is not captured by the current domain theory for fluids. In the analogous domain of electronics, it is reasonable for

Solution Phase	#Nodes	#Asns	#Envs	#Clauses	#Solutions	Time (sec)
Model Instantiation	1158	76	97	1191	—	7.38
Cand. Generation	0	0	148	2	4	7.17
Abduction Setup	1513	189	191	1610	—	17.27
Abduction (first)	5654	195	3508	6251	2	106.83
Abduction (rest)	2537	101	3012	2782	14	338.72
Total	10862	561	6956	11834	19	460.10

Table 6.5: Performance Summary for the Clog Example

a current path to become disconnected without other paths being generated. This type of behavior is specific to each domain, and requires knowledge of the mechanisms by which failures occur; this issue is discussed further in Section 7.3.

The performance on this example is summarized in Table 6.5. Note that the singleton candidates involving the pipe are found very quickly compared to the less plausible candidates which require abductive completion. Abduction on the two conjunctive goals is quite costly, and should be performed only after the more probable singleton candidates have been eliminated through additional probes or other experimentation.

The combinatorics of the abduction phase results from the independence of the two conjuncts of the top-level goal. There is in practice no need to compute the sixteen combinations of the four explanations for each container. Instead, each violated closed-world assumption may be explained in isolation. For this example, this cuts in half the number of explanations generated, and reduces the run-time for abduction even more. For more complex examples involving more conjunctive goals, the savings would be even greater.

6.6 A Partially Clogged Pipe

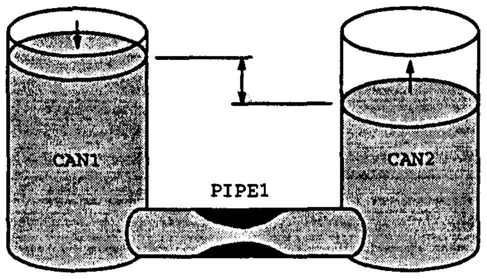
The previous example involved a total blockage of a pipe. Such a failure could be caused by a gradual accumulation of deposits on the pipe's inner wall. If so, it would be beneficial to detect and respond to the blockage before the flow is completely shut off. A partial blockage will exhibit a decreased flow rate for a given pressure difference, as compared with the unblocked pipe. This qualitative knowledge allows us to diagnose the slow flow as a partial blockage in the pipe, among other possibilities.

This example involves a drift failure which leads to symptoms called *magnitude deviations.* Magnitude deviations are not detectable using a purely qualitative model, but a human observer or a numerical simulation could detect such a discrepancy and provide it to the system in qualitative terms. Given these symptoms, a qualitative model is sufficient to trace the perturbation back to its source. This section demonstrates how this is done in PDE.

Figure 6.25 shows the same two-container fluid system as described in the previous example, but with a different set of observations. The higher of the two liquids is observed to be falling but more slowly than expected, while the lower liquid level is slowly rising. These observations are symptomatic, due to the drift assumption mechanism described in Section 3.6.5. The default zero-drift assumptions are included in the set of base assumptions eligible for retraction.

Scenario Description:

```
(Container CAN1)  (Pipe PIPE1)  (Container CAN2)
(Connection PIPE1 (bottom CAN1) (bottom CAN2))
```



Observations:

```
(Positive (volume (liquid_in CAN1)))
(Positive (volume (liquid_in CAN2)))
(> (level (liquid_in CAN1))
   (level (liquid_in CAN2)))
(Slowly_Decreasing
   (level (liquid_in CAN1)))
(Slowly_Increasing
   (level (liquid_in CAN2)))
(Q_OK (level (liquid_in CAN1)))
(Q_OK (level (liquid_in CAN2)))
```

Figure 6.25: A Partially-Clogged Pipe Connecting Two Containers

```
<E-1682*> - <E-95>:
  (Positive (mass (C_S WATER LIQUID CAN1)))
  (Positive (mass (C_S WATER LIQUID CAN2)))
  (Positive (volume (liquid_in CAN1)))
  (Positive (volume (liquid_in CAN2)))
  (Decreasing (level (liquid_in CAN1)))
  (Increasing (level (liquid_in CAN2)))
  (Fast (level (liquid_in CAN1)))
  (Slow (level (liquid_in CAN2)))
  (Q_OK (conductance PIPE1))
  (Q_OK (density (liquid_in CAN1)))
  (Q_OK (level (liquid_in CAN1)))
  (Q_OK (level (liquid_in CAN2)))
  (Q_OK (volume CAN1))
  (Q_OK (volume CAN2))
  (> (level (liquid_in CAN1))  (level (liquid_in CAN2)))
  (< (height (bottom CAN1))  (level (liquid_in CAN1)))
  (< (height (bottom CAN2))  (level (liquid_in CAN2)))
```

Figure 6.26: Base Environment for the Partially-Clogged Pipe

```
<E-429> - <E-1682*>:
  (Low (conductance PIPE1))

<E-431> - <E-1682*>:
  (Low (density (liquid_in CAN1)))

<E-436> - <E-1682*>:
  (CWA 52: (Affected (delta (net-influence (mass (C-S WATER LIQUID CAN1)))) +))
  (CWA 55: (Affected (delta (net-influence (mass (C-S WATER LIQUID CAN2)))) -))
```

Figure 6.27: Three Candidates for the Partially Clogged Pipe Example

6.6.1 Candidate Generation

The candidate generator in PDE detects a conflict involving the observations and the nominal values for the liquid levels, the pipe's conductance, and the liquid's density. The lower-than-expected flow rate is explained if the pipe's conductance or the liquid's density is low. The density of water is roughly constant;[9] thus if we are certain that no one has substituted molasses for water in the fluid system, then we are left with a single explanation: the conductance of the pipe has decreased. A detailed model of pipes relating conductance to length and cross-sectional area would allow us to further refine our hypothesis to a restriction in (or stretching of) the pipe; this is beyond the current resolution of the domain theory.

Note that the explanation found for the partial clog does not generalize to explain the previous example of a total clog. This is because the domain theory requires that pipes have a positive conductance. The conductance is not allowed to drift to zero, so a drift alone cannot explain the total clog. If the domain theory were relaxed to allow a zero conductance, then drift of the conductance to zero would be a valid diagnosis. Whether a total clog is modeled as a discrete removal of the pipe or as a continuous drift of the conductance to zero depends on the whims of the individual model builder.

6.6.2 Abductive Backchaining

A sum (such as a net influence) can be high either because one of its contributing members is high, or because it has additional, unknown members which make it higher than expected. This latter case is detected as a violated closed-world assumption representing the elements of the sum, as occurs in the last candidate found. This candidate is similar to the one found for the previous example, where the liquid levels had stopped completely. Figure 6.28 shows the top portion of the justification tree constructed for the abductive goal. Note that the justification tree contains as subgoals the top-level goals of the previous example, and in fact the same abductive explanations (listed in Figure 6.24) are found here.

[9] Ignoring minor variations with temperature.

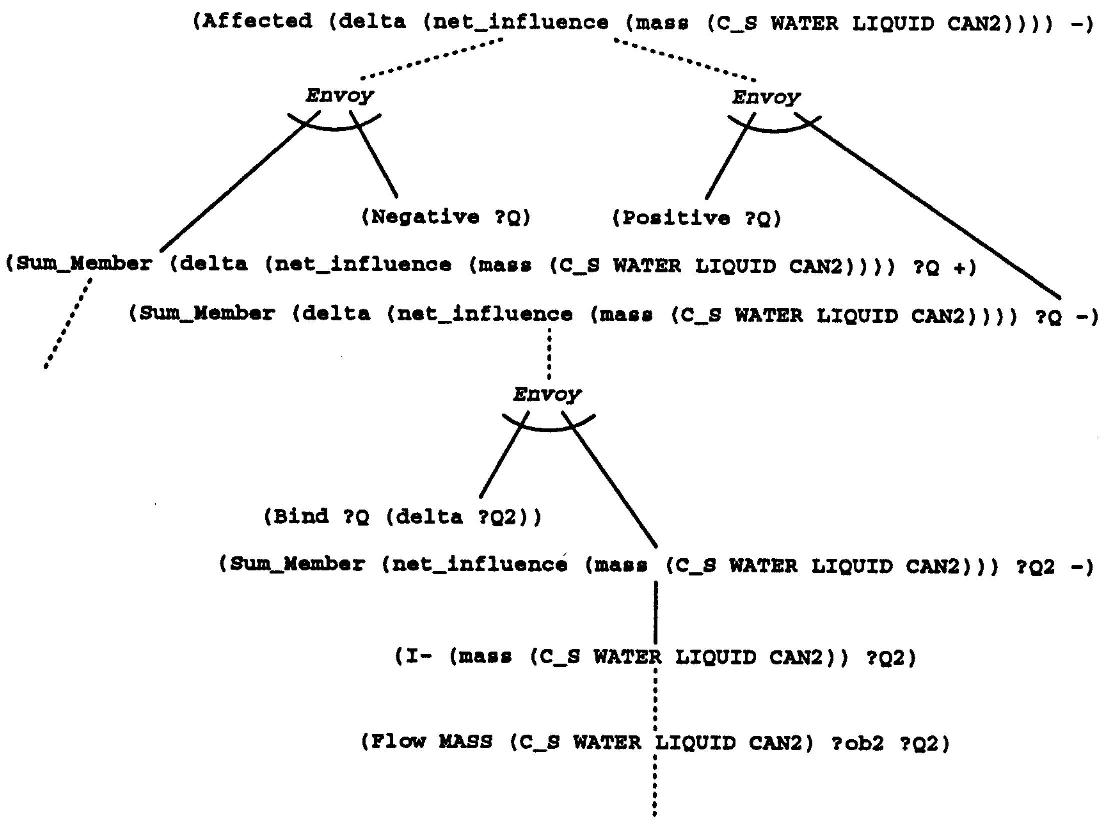

(Affected (delta (net_influence (mass (C_S WATER LIQUID CAN2)))) -)
Envoy
Envoy
(Negative ?Q)
(Positive ?Q)
(Sum_Member (delta (net_influence (mass (C_S WATER LIQUID CAN2)))) ?Q +)
(Sum_Member (delta (net_influence (mass (C_S WATER LIQUID CAN2)))) ?Q -)
Envoy
(Bind ?Q (delta ?Q2))
(Sum_Member (net_influence (mass (C_S WATER LIQUID CAN2))) ?Q2 -)
(I- (mass (C_S WATER LIQUID CAN2)) ?Q2)
(Flow MASS (C_S WATER LIQUID CAN2) ?ob2 ?Q2)

Figure 6.28: Abductive Justifications for a Slow Level in CAN2.

6.6.3 Discussion

This example demonstrates the diagnosis of a drift failure, as described in Section 3.6. While qualitative models are unable to detect such failures, they may be used to diagnose them once detected. The diagnosis of a drift failure requires establishing a perspective from which to reason. In this example, the observed levels provide the perspective, and the slowly-changing levels are the symptoms. If we do not include the observations that the liquid levels are "ok" (and give them suitably-high probabilities), then we can explain the slow flow by assuming that the level in CAN1 is low or that the level in CAN2 is high. This notion of perspective is different from the one introduced by Weld [77], which was used to compare two behaviors over an interval of time, to determine relative duration, for example. Still, the drift mechanism may be viewed as implimenting a simple form of Weld's differential qualitative analysis, but lacking the machinery needed for temporal reasoning.

The explanations found by abduction demonstrate that it is possible for a magnitude deviation to be caused by other than a drift failure. Unknown processes can act to reduce the effects of the flow process, thereby giving the same symptoms as a partially-clogged pipe. The ability in PDE to explain a magnitude deviation as a violated closed-world assumption enables it to then abductively search for such processes and their underlying structure.

Solution Phase	#Nodes	#Asns	#Envs	#Clauses	#Solutions	Time (sec)
Model Instantiation	1684	118	186	1991	—	14.83
Cand. Generation	0	0	212	1	3	20.43
Abduction Setup	1611	195	197	1656	—	18.67
Abduction (first)	5713	198	3616	6320	2	146.40
Abduction (rest)	2642	112	3095	2902	14	524.86
Total	11650	623	7306	12870	18	725.19

Table 6.6: Performance Summary for the Partial Clog Example

Table 6.6 lists the run times and other performance statistics for this example. The inclusion of the drift assumptions and associated mechanisms nearly doubles the representation within the ATMS, and significantly increases the time required to generate initial candidates, as compared with the previous example. The time required to perform the abduction phase is only slightly slower than for the previous example.

6.7 A Seven-Container Fluid System

This example involves a more complicated fluid system than the previous ones; it is included to demonstrate how the performance of PDE scales with increased problem complexity. The fluid system shown in Figure 6.29 consists of seven containers, one of which (CAN7) is connected to each of the other six by either a pipe or a pump. Three of the pipes have valves which are believed to be closed. Only one of the two pumps is believed to be on. The model predicts that the level of liquid in CAN7 should be falling, which conflicts with the observation that the level is rising.

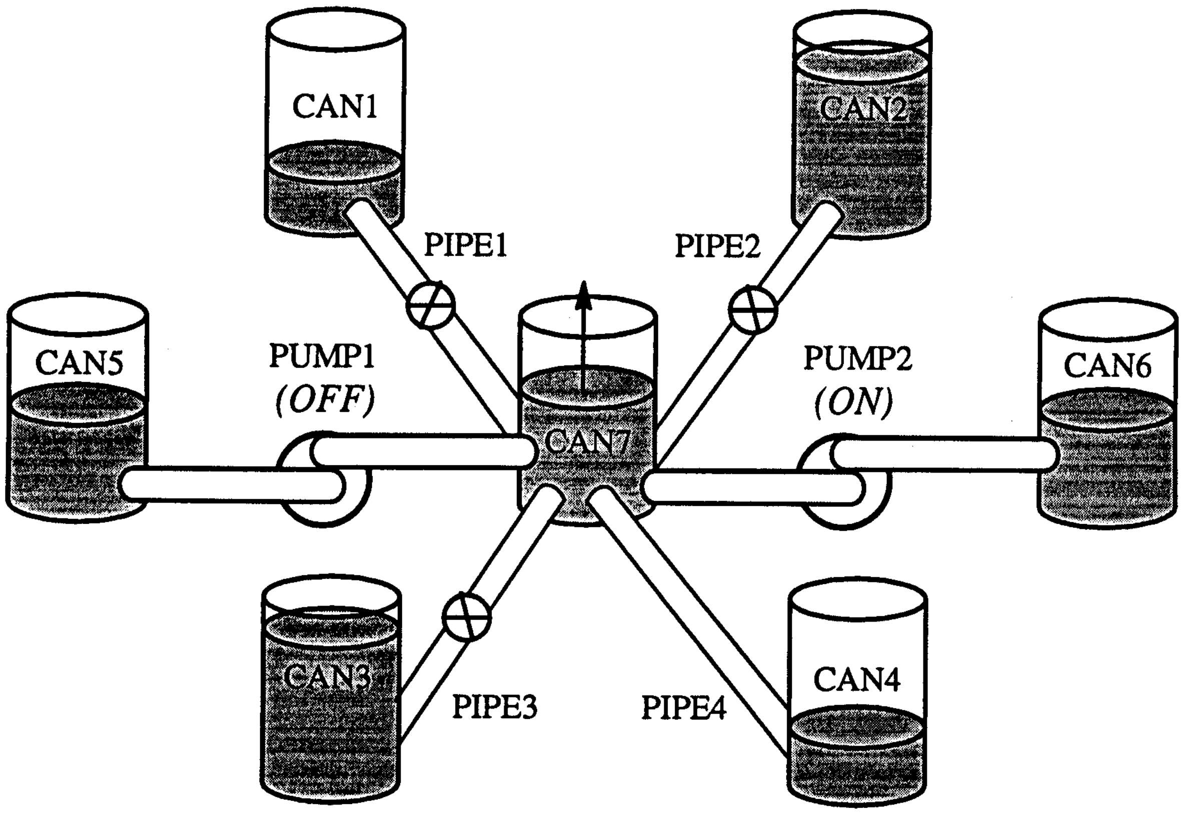


Figure 6.29: A Seven-Container Fluid System

```
(Container CAN1)  (Container CAN2)
(Container CAN3)  (Container CAN4)
(Container CAN5)  (Container CAN6)
(Container CAN7)
(Pipe PIPE1)      (Closed (valve_in PIPE1))
(Pipe PIPE2)      (Closed (valve_in PIPE2))
(Pipe PIPE3)      (Closed (valve_in PIPE3))
(Pipe PIPE4)
(Pump PUMP1)      (Off PUMP1)
(Pump PUMP2)      (On  PUMP2)
(Connection PIPE1 (bottom CAN1) (bottom CAN7))
(Connection PIPE2 (bottom CAN2) (bottom CAN7))
(Connection PIPE3 (bottom CAN3) (bottom CAN7))
(Connection PIPE4 (bottom CAN4) (bottom CAN7))
(Connection PUMP1 (bottom CAN5) (bottom CAN7))
(Connection PUMP2 (bottom CAN7) (bottom CAN6))
```

Figure 6.30: Scenario Description for the Seven-Container Example

```
(Positive (volume (liquid_in CAN1)))
(Positive (volume (liquid_in CAN2)))
(Positive (volume (liquid_in CAN3)))
(Positive (volume (liquid_in CAN4)))
(Positive (volume (liquid_in CAN5)))
(Positive (volume (liquid_in CAN6)))
(Positive (volume (liquid_in CAN7)))
(> (level (liquid_in CAN7)) (level (liquid_in CAN1)))
(< (level (liquid_in CAN7)) (level (liquid_in CAN2)))
(< (level (liquid_in CAN7)) (level (liquid_in CAN3)))
(> (level (liquid_in CAN7)) (level (liquid_in CAN4)))
(Decreasing (level (liquid_in CAN7)))
```

Figure 6.31: Observations for the Seven-Container Example

6.7.1 Candidate Generation

Figure 6.32 shows the base environment for this example, excluding the closed-world assumptions, which are too numerous to mention (183 in all). This environment is contradictory, due to the conflicting beliefs regarding the direction of the liquid level in CAN7. The minimal retraction algorithm is given this environment, and finds the four minimal candidates shown in Figure 6.33.

The first two candidates suggest that one of the valves in PIPE2 or PIPE3 might be open, contrary to original indications. This would enable a flow of liquid from CAN2 or CAN3 to CAN7, which could account for the rising level there. The third candidate suggests that PUMP1 could be on, and thus cause the level in CAN7 to rise. Note that opening the valve in PIPE1 would not produce the observed symptoms, as it would only lead to additional flow out of CAN7. Similarly, turning off PUMP2 would not cause the level in CAN7 to rise, but merely to stop falling so fast. The last candidate indicates that some additional process(es) could explain the rising fluid level; this possibility is represented by the violated closed-world assumption for the positive influences on the contained liquid in CAN7.

6.7.2 Abductive Backchaining

The violated closed-world assumption in the last candidate in Figure 6.33 is similar to the one found for the clogged pipe example, and the abductive search for an explanation is virtually identical. GA3 is given the top-level goal shown at the top of Figure 6.34, and finds the four explanations indicated. The first explanation is that the liquid level is rising due to condensation within CAN7. This explanation is valid given the qualitative symptoms, even though intuitively condensation is too slow a process to yield an observable rise in a liquid's level—especially when it has to compete with PUMP2 which is presumably still pumping liquid out of CAN7.[10] The second candidate involves an unknown pump whose output is connected to CAN7. The last

[10]Order of magnitude reasoning [59] could be applied to filter such explanations; this is an important area for future research.

```
Environment <E-356*>:
  (Container CAN1)  (Container CAN2)
  (Container CAN3)  (Container CAN4)
  (Container CAN5)  (Container CAN6)
  (Container CAN7)
  (Pipe PIPE1)      (Pipe PIPE2)
  (Pipe PIPE3)      (Pipe PIPE4)
  (Pump PUMP1)      (Pump PUMP2)
  (Off PUMP1)       (On  PUMP2)
  (Closed (valve_in PIPE1))
  (Closed (valve_in PIPE2))
  (Closed (valve_in PIPE3))
  (Connection PIPE1 (bottom CAN1) (bottom CAN7))
  (Connection PIPE2 (bottom CAN2) (bottom CAN7))
  (Connection PIPE3 (bottom CAN3) (bottom CAN7))
  (Connection PIPE4 (bottom CAN4) (bottom CAN7))
  (Connection PUMP1 (bottom CAN5) (bottom CAN7))
  (Connection PUMP2 (bottom CAN7) (bottom CAN6))
  (Positive (mass (C_S WATER LIQUID CAN1)))
  (Positive (mass (C_S WATER LIQUID CAN2)))
  (Positive (mass (C_S WATER LIQUID CAN3)))
  (Positive (mass (C_S WATER LIQUID CAN4)))
  (Positive (mass (C_S WATER LIQUID CAN5)))
  (Positive (mass (C_S WATER LIQUID CAN6)))
  (Positive (mass (C_S WATER LIQUID CAN7)))
  (Positive (volume (liquid_in CAN1)))
  (Positive (volume (liquid_in CAN2)))
  (Positive (volume (liquid_in CAN3)))
  (Positive (volume (liquid_in CAN4)))
  (Positive (volume (liquid_in CAN5)))
  (Positive (volume (liquid_in CAN6)))
  (Positive (volume (liquid_in CAN7)))
  (Decreasing (level (liquid_in CAN7)))
  (>  (level (liquid_in CAN7))  (level (liquid_in CAN1)))
  (<  (level (liquid_in CAN7))  (level (liquid_in CAN2)))
  (<  (level (liquid_in CAN7))  (level (liquid_in CAN3)))
  (>  (level (liquid_in CAN7))  (level (liquid_in CAN4)))
```

Figure 6.32: Base Environment for the Seven-Container Example

```
<E-1113> - <E-356*>:
  (Open (valve_in PIPE2))

<E-2990> - <E-356*>:
  (Open (valve_in PIPE3))

<E-3053> - <E-356*>:
  (On PUMP1)

<E-3054> - <E-356*>:
  (not (CWA 71: (Affected (net_influence (mass (C_S WATER LIQUID CAN7))) +)))
```

Figure 6.33: Four Candidates for the Seven-Container Example

Solution Phase	#Nodes	#Asns	#Envs	#Clauses	#Solutions	Time (sec)
Model Instantiation	3480	221	452	3733	—	33.15
Cand. Generation	0	0	2602	17	4	400.50
Abduction Setup	1057	91	129	1146	—	16.84
Abduction (first)	2872	96	1403	3285	2	19.56
Abduction (rest)	1342	53	1021	1505	2	97.14
Total	8751	461	5607	9686	7	567.19

Table 6.7: Performance Summary for the Seven-Container Example

two candidates suggest that CAN7 has an additional pipe connected to it, and that liquid is flowing through this pipe from some unknown source into CAN7. The two environments differ in the direction of connectivity of the pipe.

6.7.3 Discussion

Table 6.7 summarizes the performance on this example. Note that this example requires significantly more time for candidate generation than the previous examples based on the two-container fluid system. It also involves several times more infrastructure in the ATMS. However, by comparing these results with those of the previous examples, we see that the cost of performing the abductive search is only slightly affected by the complexity of the surrounding system.

In this example, all of the pipes and pumps participate in the inference that the level in CAN7 is not rising, so the traditional consistency-based approach based on the "constraint suspension" view of diagnosis would have to find candidates involving all of them. Allowing for unknown failure modes means allowing for a pipe to fail in such a way that it moves water uphill. Such an approach is too weak to say anything interesting about examples such as this one. On the other hand, the process-based approach replaces unknown failure modes with the assumption

```
;;;; Abductive Goal:
(Affected (net_influence (mass (C_S WATER LIQUID CAN7))) +)

;;;; Abductive Explanations:
Environment <E-4742>:
  (Container CAN7)
  (Physob SK-SRC-17)
  (Heat_Connection SK-HPATH-16 SK-SRC-17 (C_S WATER LIQUID CAN7))
  (Positive (Generation_Rate LIQUID CAN7))
  (<= (temperature (C_S WATER GAS CAN7) ABS)
      (temperature (boil (C_S WATER GAS CAN7)) ABS))
  (>  (temperature (C_S WATER GAS CAN7) ABS)
      (temperature SK-SRC-17 ABS))
  (Positive (heat SK-SRC-17))

Environment <E-4985>:
  (Container SK-C-11)
  (Fluid_Pump SK-PATH-14)
  (Connection SK-PATH-14 (SK-N-12 SK-C-11) (SK-N-13 CAN7))
  (On SK-PATH-14)
  (Portal (SK-N-12 SK-C-11))
  (Positive (flow_rate SK-PATH-14 (SK-N-12 SK-C-11) (SK-N-13 CAN7) WATER LIQUID))
  (<  (height (SK-N-13 SK-C-11)) (height (liquid_in SK-C-11)))

Environment <E-5141>:
  (Container SK-C-11)
  (Pipe SK-PATH-14)
  (Connection SK-PATH-14 (SK-N-12 SK-C-11) (SK-N-13 CAN7))
  (Portal (SK-N-13 SK-C-11))
  (not (Consider (fluid_conductance SK-PATH-14)))
  (Aligned SK-PATH-14)
  (>  (pressure (SK-N-12 SK-C-11) ABS) (pressure (SK-N-13 CAN7) ABS))
  (Positive (flow_rate SK-PATH-14 (SK-N-12 SK-C-11) (SK-N-13 CAN7) WATER LIQUID))
  (<  (height (SK-N-13 SK-C-11)) (height (liquid_in SK-C-11))))

Environment <E-5327>:
  (Container SK-C-11)
  (Pipe SK-PATH-24)
  (Connection SK-PATH-24 (SK-N-21 CAN7) (SK-N-23 SK-C-22))
  (Portal (SK-N-21 SK-C-11))
  (not (Consider (fluid_conductance SK-PATH-24)))
  (Aligned SK-PATH-24)
  (>  (pressure (SK-N-23 SK-C-22) ABS) (pressure (SK-N-21 CAN7) ABS))
  (Negative (flow_rate SK-PATH-24 (SK-N-21 CAN7) (SK-N-23 SK-C-22) WATER LIQUID))
  (<  (height (SK-N-21 SK-C-11)) (height (liquid_in SK-C-11)))
```

Figure 6.34: Four Abductive Explanations for the Seven-Container Example

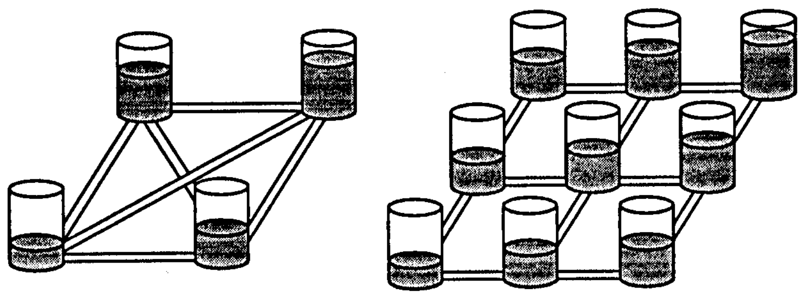


Figure 6.35: Two Fluid System Configurations

that a component is either fully-functional or it has been replaced with some other (possibly null) assembly of components from the known types. This allows a much stronger sense of explanation than the traditional consistency-based approach, as it rules out candidate models which do not have valid completions within the given domain theory.

6.8 Scaling Up in the Fluids Domain

This section describes the diagnosis of two families of fluid systems of varying size. The first consists of (some number) N containers of liquid, totally connected to each other by pipes. The second consists of a square NxN Manhattan grid of containers, with each container connected to its four immediate neighbors.[11] Each pipe contains a valve which is believed to be closed. Figure 6.35 shows an example of each class of system.

Fluid levels in the totally-connected systems are totally ordered in correspondence with the container's number, with the first container having the lowest level and the Nth container having the highest. In the Manhattan grid systems, levels are partially ordered, increasing along rows and columns, so that the lowest level is at (0,0) and the highest level is at $(N-1,N-1)$. In each system, the "middle" container (designated C_f) is observed to be falling, contrary to the model's prediction that it should be constant. This example is a generalized version of the leak example presented in Section 6.4. Various sizes of each configuration of fluid system are diagnosed to demonstrate how the performance of PDE scales with increasing problem complexity.

6.8.1 Candidate Generation

Each of the two systems is diagnosed for various values of N, where N is the number of containers in the totally-connected system, and the number of rows (and columns) in the Manhattan grid. In each case, the model predicts that the fluid level in C_f should be constant, which conflicts with the observation that it is decreasing. The candidate generator is given the inconsistent base environment, consisting of the scenario description, observations, modeling

[11] Containers along an outer edge are connected to three other containers, and the four corner containers are only connected to two.

```
<Env-886> - <Env-527*>:
  (Open (valve_in PIPE0-2))

<Env-931> - <Env-527*>:
  (Open (valve_in PIPE1-2))

<Env-1008> - <Env-527*>:
  (not (CWA 43: (Affected (net_influence (mass (C_S WATER LIQUID CAN2))) -)))
```

Figure 6.36: Three Candidates for Five Totally-Connected Containers

```
<Env-6542> - <Env-2237*>:
  (Open (valve_in PIPE2-1H))

<Env-6563> - <Env-2237*>:
  (Open (valve_in PIPE1-2V))

<Env-6599> - <Env-2237*>:
  (not (CWA 68: (Affected (net_influence (mass (C_S WATER LIQUID CAN2-2))) -)))
```

Figure 6.37: Three Candidates for the 5x5 Manhattan Grid

and closed-world assumptions, and returns a set of environments representing the minimal retractions from the base required to restore consistency. Most of these consist of individual valves connected to C_f which are unexpectedly open. The number of such candidates is equal to the number of pipes connecting C_f to a lower fluid level.[12] For the Manhattan grid systems, this is two; for the totally-connected systems, it is $\lceil (N-1)/2 \rceil$. In addition to the candidates in which a valve has opened, there is one additional candidate containing a violated closed-world assumption for the set of negative influences on the amount of water in C_f. This candidate covers the possibility of a leak or other unknown process which could be removing water from C_f.

Figures 6.36 and 6.37 show the candidates found for the totally-connected and Manhattan grid configurations, respectively, for $N = 5$. In both cases, three candidates are found: two involving an open valve and the third containing a violated closed-world assumption. For larger values of N, the totally-connected system yields more candidates.

[12]Opening a valve to a higher level does not explain why the level in C_f is dropping.

Fluid System Size			Time (sec)			
#Containers	#Pipes	#Candidates	Setup	Cands	Abduct	Total
2	1	2	8.38	3.25	27.63	39.26
3	3	2	16.98	8.57	29.52	55.07
4	6	3	32.07	24.38	33.47	89.92
5	10	3	48.00	44.08	39.80	131.88
6	15	4	79.88	90.10	56.58	226.56
7	21	4	139.26	175.25	82.67	397.18
8	28	5	276.17	335.10	134.49	745.76

Table 6.8: Performance Summary for N Totally-Connected Containers

Fluid System Size				Time (sec)			
N	#Containers	#Pipes	#Candidates	Setup	Cands	Abduct	Total
2	4	4	3	19.25	10.92	31.64	61.81
3	9	12	3	58.17	54.57	39.29	152.03
4	16	24	3	182.83	174.18	48.15	405.16
5	25	40	3	576.35	436.52	77.90	1090.77
6	36	60	3	1643.80	1226.47	143.63	3013.90

Table 6.9: Performance Summary for the NxN Grid of Containers

6.8.2 Abductive Backchaining

The last candidate generated for each type of system contains a violated closed-world assumption, requiring further explanation. The violated closed-world assumption is the same for both system configurations and for all values of N, and is identical to the one found for the leak example in Section 6.4. This closed-world assumption provides the goal for abductive search, which finds the same four explanations as shown in Figure 6.19. Theoretically the cost of performing this search should be independent of N, but run-times reflect some overhead associated with the total number of ATMS assumptions and environments. Much of this overhead is probably avoidable; it is up to future versions of PDE to determine the extent to which this is true.

6.8.3 Discussion

Table 6.8 summarizes the performance of PDE for various numbers of totally-connected containers. The run-times for the candidate generator appear to be approximately $O(N^4)$, or $O(A^2)$ where A is the number of assumptions. Again this can be traced to the inefficiency of the whittling approach used for finding minimal nogoods.

Table 6.8 summarizes performance for the Manhattan grid configuration. The number of assumptions here are also $O(N^2)$, and the run-times are comparable to the totally-connected case. For a given N the Manhattan grid has four times as many pipes (and N times as many containers) as the totally-connected configuration, but fewer pipes connected to C_f and thus fewer candidates (for $N > 5$).

The totally-connected and Manhattan grid configurations are admittedly ad hoc; still, the complexity of these systems rivals that of a typical fluid system found in a power plant or chemical processing facility, for example. The run times, while slower than desired, are approaching the point of viability for real-time monitoring and diagnosis of substantial engineered fluid systems.

6.9 Summary

The examples in this chapter demonstrate the two-tiered approach which characterizes process-based diagnosis. In the first step, the traditional consistency-based approach is used to identify minimal departures from the original model which are consistent with observations. In the second step, abductive backchaining is used to explain any violated closed-world assumptions from the first step. This unique combination of diagnostic approaches yields greater efficiency and inferential power than either approach alone.

The traditional consistency-based approach is too weak in that it finds too many candidates. Until now, explicit fault models were required to rule out physically impossible candidates. The assumption of a closed domain theory allows us to search for structural explanations for a failure, and to rule out candidates when no such explanations exist. Any new structure created by the failure is detected as closed-world assumption violations, which are then explained abductively.

Similarly, a pure abductive approach would be both less efficient and less powerful than the combined approach. For many of the examples in this chapter, abductive search applied directly to the observations would fail to find any explanations, due to the inherent ambiguity of the qualitative model. By backchaining on an intermediate goal between behavior and structure, abduction is both more efficient and more likely to find an explanation. In addition to providing the goal for abduction, the candidate generator provides a consistent context in which to focus the abductive search and further reduce its potentially-high cost. The ability to diagnose violated closed-world assumptions opens up a new world of possibilities to model-based diagnosis, by allowing us to consider additions to as well as deletions from the set of structural beliefs. This ability to efficiently explore the space of scenario descriptions is one of the unique contributions of this work.

Most of the examples of this chapter are drawn from the domain of fluids and thermodynamics; still, the approach itself is quite general and is in no way limited to a single domain. The inclusion of the first two examples from the electronics domain demonstrate this. It is recognized that the greatest challenges in model-based diagnosis lie in developing adequate models. While we make no strong claims as to the adequacy of the current domain theories, we do claim that this framework is more conducive to model development than the conventional consistency-based approach combined with explicit fault models.

In particular, we claim that the model-closing mechanism required for compositional, process-centered models provides a more natural way to express the constraints for a domain than either explicit fault models [75] or physical impossibility axioms [75]. Rather that enumerating all the possible ways that each component can fail, or all behaviors which are impossible, a closed domain theory specifies the possible behaviors within the domain, and is assumed to be complete in doing so. This is a natural extension of default reasoning, an ingredient in any approach to model-based diagnosis.

The performance of PDE on the examples in this chapter demonstrates both the solvency of the overall approach, and the potential viability of the implementation as a serious diagnostic tool. The current implementation was designed as a research tool, and is not sufficiently bulletproofed for general use. Still, these results suggest that the process-based approach and its implementation constitute significant and novel contributions to the mix of tools available for automated diagnosis.

Chapter 7

Conclusions

This chapter summarizes the major claims and contributions of process-based diagnosis, shows its relationship to other research areas, discusses some of its limitations and identifies areas for future research.

7.1 Review of Process-Based Diagnosis

This thesis presents a novel approach to diagnosis, based on the integration of consistency-based and abductive-based diagnostic techniques. This integration was a natural consequence of applying traditional model-based diagnosis to the process-centered models of QP theory. The causal nature of process-centered models demands an assumption that all causes have been enumerated; i.e., that the model is complete. This closed-world assumption is often violated by a failure, and so must be represented explicitly so that it may be retracted. Explaining a closed-world assumption violation requires an abductive search for the possible causes for an unexpected behavior.

Process-based diagnosis attempts to achieve a deeper understanding of a failed device, through the application of first-principles qualitative models. Unlike consistency-based diagnosis which weakens the model until the conflict disappears, process-based diagnosis searches for a modified structural description of a broken device whose model explains the observed symptoms.

7.1.1 Review of Claims

We reiterate here the major claims of process-based diagnosis, initially stated in Chapter 1.

Failures reside within the model, not the device. A broken device continues to obey the natural laws; it is our model of it which has failed to keep pace with changes within the device. This view is widely accepted among model-based diagnosis researchers; however, until now it has not been taken to its logical conclusion. Under this view, diagnosis is seen as an attempt to understand and repair a failed model.

Diagnosis should explain failures. For an automated diagnosis system to be considered intelligent, it must go beyond localizing a failure to one or more components; it must seek a deeper understanding of the structure of the failed device. Devising a structural model for the device after failure can help to detect and eliminate physically impossible candidates, and to better understand the nature of the failure and its underlying cause.

Many failures result in a modified process structure. A broken device may be understood in terms of its underlying active processes. A failure may activate new processes or deactivate existing ones. For example, a leak in a fluid system may be seen as an instance of a fluid flow process. We require no explicit fault model for leaks, as long as fluid flows are already part of the domain knowledge.

Some kinds of knowledge are beyond suspicion. The process-centered models used in this research are composed from a general domain theory and a specific structural description for a device. We assume that the domain theory is correct (i.e., both sound and complete), and remains applicable to the broken device. In particular, we assume that a failed component may be modeled as a (possibly null) assembly of working components which are modeled within the domain theory. This assumption requires that the domain theory be sufficiently broad to cover a device both before and after any conceivable failure. As demanding as this may seem, we claim that it is more accommodating than the explicit enumeration of all possible failure modes for each component. Given this assumption, diagnosis is viewed as a search for a modified structural description of the device after failure.

Process-centered models support efficient search. Unlike the flat associational models of early diagnostic approaches, process-centered models provide a rich representation containing many intermediate layers between the physical structure of a device and its predicted behavior. By initially focusing on the active process structure rather than the underlying physical structure of a failed device, the search space is greatly reduced. This search is also focused by knowledge of which quantities are changing unexpectedly, given that each quantity can only be influenced by a few types of processes.

Structure added by a failure violates closed-world assumptions. Explicit representation of closed-world assumptions allows us to question the belief that what we know about is all there is. If a failure introduces new structure, it may be initially detected as a closed-world assumption violation. During candidate generation, closed-world assumptions are handled like any other defeasible assumption, and so may be retracted to eliminate an inconsistency. Candidates containing a violated closed-world assumption may be further refined using abductive search.

7.1.2 Review of Contributions

Process-based diagnosis is a general approach to understanding a failed device; it is applicable to any device in any domain for which a causal model exists. In particular, the process-centered qualitative models of QP theory may be used to explain symptoms without relying on explicit fault models, thereby increasing robustness for novel faults. By assuming that the domain theory is both sound and complete, diagnosis is transformed into a search for a modified scenario description whose model is consistent with the observed symptoms. This research has made specific contributions in the following areas.

7.1.2.1 A Unique Combination of Two Diagnostic Approaches

This research combines consistency-based diagnosis with abductive backchaining. The consistency based approach is used initially to search for minimal departures from the original model, consistent with the observed symptoms. This provides the focus and starting point for an abductive search for missing fragments of the model. By combining the strengths of the two approaches, we effectively divide up the problem so as to increase both inferential power and reasoning efficiency.

The traditional consistency-based approach based on models of correct behavior only is often unable to rule out physically impossible candidates. Explicit fault models and physical impossibility axioms are the two most commonly prescribed remedies. In process-based diagnosis, the assumption of a closed domain theory allows us to search for structural explanations for a failure, and to rule out candidates when no such explanations exist. Any new structural entities or connections created by the failure are detected as violated closed-world assumptions, which are then explained abductively.

On the other hand, a pure abductive approach would be both less efficient and less powerful than the combined approach. Because of the ambiguity inherent in qualitative models, abductive backchaining can fail to find an explanation for an observed symptom. Violated closed-world assumptions provide an intermediate goal between behavior and structure, making abduction both more efficient and more likely to find an explanation. The candidate generator also provides a consistent context in which to focus the abductive search and further reduce its potentially-high cost.

7.1.2.2 Modeling

The modeling language presented in Chapter 3 allows for the concise expression of domain theories and domain-independent constraints. This language has evolved from the language used in QPE, simplifying its syntax while extending its expressiveness. The inclusion of translation rules allows the model builder the freedom to choose the most natural representation without sacrificing performance.

The models used for diagnosis need not be designed specifically with that task in mind. One of the benefits of a composable model is reusability; this applies not only to different scenarios but also to different applications. For example, the domain theory for thermodynamics used in this research was originally developed for the purpose of forward simulation. Reusability makes it possible to amortize the cost of developing a domain theory across many applications, thus justifying greater expenditure in domain theory development.

7.1.2.3 Utilizing Incontrovertible Knowledge

In model-based reasoning, the model is everything; it represents the extent of our knowledge about the modeled device and the world in which it exists. Thus to question the validity of the model is to challenge everything we know. In traditional consistency-based diagnosis, as characterized by the constraint-suspension approach, no part of the model is beyond suspicion; any subset of the model could be wrong. Having no certain knowledge to fall back on, the consistency-based approach can only whittle away at the original model, weakening it until it no longer makes false predictions.

The approach taken here is to hold some knowledge in reserve, as being beyond question. Having a foundation of certain knowledge gives us a place to stand as we disassemble and reassemble the uncertain knowledge in search of a valid model. This distinction between certain and uncertain knowledge partitions the model into two distinct constituents. Interestingly, these correspond exactly to the two constituents of the process-centered models of QP theory—namely, the domain theory and the scenario description.[1] A carefully crafted domain theory provides a solid foundation from which to explore the space of possible scenario descriptions.

7.1.2.4 Closed-World Assumptions

The assumption that a domain theory is complete—together with the causal nature of the process-centered models of QP theory—allows us to infer that an effect is false whenever all of its known causes are absent. The explicit use of closed-world assumptions in this research provides a shorthand representation for other possible causes which exist within the domain theory but have not yet been instantiated for the current model. If retracting a closed-world assumption eliminates an inconsistency, then these other causes may be instantiated using abductive backchaining. The ability to diagnose violated closed-world assumptions allows us to consider additions to as well as deletions from the set of structural beliefs. This ability to efficiently explore the space of possible scenario descriptions is one of the unique contributions of this work.

7.1.2.5 Candidate Generation

One of the contributions of this research is the minimal retraction algorithm supporting the candidate generator. Like many problems in default reasoning, diagnosis involves finding minimal retractions to a set of default beliefs to resolve an inconsistency. The algorithm presented in Chapter 4 improves on Reiter's *DIAGNOSE* algorithm [68], by pruning away branches of the search tree which can only yield redundant or non-minimal candidates. This algorithm also differs in that it does not merely *remove* suspect beliefs—it *negates* them. Each component or connection is either present or absent after failure, and we want the candidate generator to consider the consequences of the absence of each entity it tries to retract.

7.1.2.6 Abductive Backchaining

Diagnosis is a form of abduction, as it involves a search for explanations of observed phenomena. This research contributes the Generalized ATMS-based Abduction Algorithm (GA3), described in Chapter 5, which is used to postulate missing structure to explain a closed-world assumption violation. GA3 overcomes the limitation of AAA [57] which precludes goal variable instantiation. Without the ability to find specific instances of a general goal, abductive backchaining would not be applicable to explaining closed-world assumption violations given the current domain theories. The introduction of binding assumptions removes this limitation, resulting in a general-purpose abduction algorithm. This algorithm has potential applications beyond diagnosis.

[1]In this context the term "scenario description" is used broadly to include the structure of the device, the observations defining its state, and the modeling assumptions which select among alternative perspectives within the domain theory.

Another feature of GA3 is the ability to take advantage of variant generalization subgoals, which already prove a more general version of a goal, but under different variable names. This eliminates much of the redundancy in the justification tree constructed by GA3, thereby improving performance. More importantly, it allows a finite structure of justifications to represent a recursive domain, and to generate in best-first order the elements of an infinite set of abductive explanations.

7.1.2.7 Abstract and Hierarchical Reasoning

The two-stage approach to diagnosis taken in this research is a way of dividing up the problem into manageable subproblems. Allowing the candidate generator to consider violated closed-world assumptions is a form of abstract reasoning. Candidates containing closed-world assumption violations represent abstract descriptions of the actual physical form of the failed device. Each such abstract candidate represents a number of possible instantiations, each requiring additional effort to identify. Eliminating an abstract candidate eliminates an entire subspace of the search space, without ever having explored that subspace in detail. This is one of the strengths of this approach.

Abstract reasoning in its more general form is made possible by the concise labels of CATMS. Hierarchical representation of a complex system allows us to think about complex subsystems as if they were single components. Because of the unique ability in CATMS to block the propagation of low-level details through an abstraction hierarchy, it can reason abstractly without becoming swamped in the details. The ability to reason at multiple levels of abstraction is a key to efficient diagnosis.

7.1.2.8 The Process-based Diagnostic Engine

PDE is the implementation of the process-based diagnostic approach. It is built atop an incremental qualitative envisioner (IQE) and a hybrid ATMS (CATMS), and includes the minimal retraction and GA3 algorithms mentioned above. The performance of PDE has been demonstrated on a number of examples, primarily from the domain of fluids and thermodynamics. The current implementation was designed as a research tool, and has not been fully optimized or bulletproofed; still, results to date suggest that the process-based approach and its implementation constitute significant and novel contributions to automated diagnosis.

7.2 Related Work

Given the volume of prior and current research in automated diagnosis, there is a vast amount of work which has influenced the direction of the research undertaken here. Most of the contributions of this research consist of extensions or refinements to previous approaches, or of combining previously separate approaches. This is, after all, how most scientific progress is made—by evolution rather than revolution. Listed here are some of the more important areas having direct relevance to this work.

7.2.1 Model-Based Diagnosis

The model-based approach to diagnosis has been in widespread use for more than a decade. Consistency-based diagnosis—the most popular approach in model-based diagnosis—provided the basis for the minimal retraction algorithm used by the candidate generator (Chapter 4). The GA3 abduction algorithm (Chapter 5) is a direct extension of the ATMS-based Abduction Algorithm (AAA) [57]. These most relevant works are described elsewhere in the thesis.

Abductive-based diagnosis has been investigated by some researchers [58, 8], but has not been widely embraced as an alternative to the consistency-based approach. Abductive-based diagnosis is more restrictive than the consistency-based approach, and uses a stronger notion of explanation in which the model for a candidate must predict the symptoms, rather than merely being consistent with them. This requires fully understanding the failures, which are typically represented using explicit fault models. Console and Torasso [8] describe these two approaches as two ends of a continuum, allowing for hybrid approaches whose models include both correct and faulty behavior, and whose candidates predict certain observations (e.g., symptoms) while merely being consistent with others. They point out that the abductive-based approach is only viable when the model is complete or nearly so; otherwise valid candidates will be missed. To allow for incompleteness, they include abducible assumptions representing anonymous causes, which may be included in the explanations generated. These correspond roughly to the negated closed-world assumptions in process-based diagnosis. The difference is that closed-world assumptions represent causes which are known to exist in the domain theory but which have not been instantiated in the existing propositional model for the given scenario.

Console and Torasso demonstrate that abduction can be seen as deduction on a completed theory [8]. If all non-abducible literals in the theory are completed by adding the appropriate implications (as described in Section 3.5), then the consistency-based approach will find exactly the same explanations as abduction. However, because most consistency-based candidate generators—including the one described in Chapter 4—apply only to the propositional portion of the model, they are unable to postulate the existence of new structure created by a failure. This is the reason that closed-world assumptions must be represented explicitly in the process-based approach—to designate the places within the propositional model where additional structure could make a difference.

This research is not the first to consider the use of process-centered models in model-based diagnosis. Throop [76] uses Kuipers' QPC [9] in his diagnosis research. His approach differs from the one taken here in that it uses linear approximations as a basis for computing an inverted model, which predicts underlying structure given observed behavior. Throop does not explicitly represent closed-world assumptions, and so could not consider candidates involving closed-world assumption violations.

7.2.2 Theory Revision

Research in theory revision and experiment planning [35, 61, 63, 64, 65] is closely related to diagnosis. By explicitly including assumptions representing the soundness and completeness of a domain theory, diagnosing a failed model may be extended to repairing an almost-correct theory for the domain. Experiment planning is similar to measurement (probe) planning and other forms of active diagnosis. These similarities suggest that each camp could benefit from the progress of the other.

7.2.3 Reconfiguration and Planning

Diagnosis attempts to understand a change which has already occurred to a device. The same techniques can be applied to find possible reconfigurations of a device which achieve a desired effect [10]. The distinction between planning and diagnosis has been primarily based on the increased temporal complexity of planning relative to diagnosis. However, failures can involve long chains of actions and reactions; for example, consider a jet turbine flying apart—the trajectory of each fragment becomes extremely important. The temporal aspects of such failures cannot always be ignored. As we attempt to diagnose more temporally complex failures, the tools developed for planning may become increasingly applicable in diagnosis, and vice versa.

7.2.4 Circumscription and Default Reasoning

Diagnosis is a form of default reasoning, so naturally developments in automated diagnosis are relevant in default reasoning and vice versa. The minimal retraction algorithm presented in Chapter 4 is a general technique for finding minimal departures from a set of default beliefs needed to restore consistency. When applied to temporal reasoning and the frame problem, minimal retractions of normal ($\neg AB()$) literals represent the most reasonable expectations or interpretations for a sequence of events. Assigning higher probabilities to earlier defaults achieves chronological minimization [72], one solution to the Yale Shooting Problem. The minimal retraction algorithm is already being used in IQE [30] to compute possible states following a discrete action, by minimizing discontinuities. The algorithm finds minimal retractions of the predictions derived from continuity, which are consistent with a known (or presumed) change. It is likely that other systems performing temporal or default reasoning could benefit from this algorithm as well.

7.3 Limitations and Future Work

Diagnosis has been an active research topic for many years, and further progress requires focusing on a portion of the overall problem. Consequently, there are many aspects of diagnosis which are not addressed by this research. In particular, this research has not been concerned with the details of knowledge acquisition (such as model development or probability assignment), nor with strategies for interacting with a failed device (such as probe/measurement planning). Neither has it dealt significantly with temporal observations and the special problems of intermittent failures. This section discusses some of the more serious limitations and open issues of this research.

7.3.1 The Importance of the Model

Process-based diagnosis depends on the availability of a well-developed domain theory capable of covering the device both before and after failure. This thesis makes no strong claims about the adequacy of the current domain theories; they are clearly inadequate for many more complicated systems. Neither does the thesis mean to belittle the importance of the model in model-based reasoning, or the difficulty in getting the model right. A great deal of effort will be required to develop accurate, detailed theories for the domains of interest in automated diagnosis. Better domain theories will inevitably come with time, and when they do, it is important that we are prepared to use them.

7.3.2 Observations Across Time

Observations are typically not limited to a single snapshot in time, but occur over some interval. Some failures may not be detectable (or diagnosable) based on observations from a single instant. Interpreting a temporal stream of observations requires finding a model whose predictions mirror the observed transitions. This ignores intermittent failures, which appear to alternate between two or more models for no apparent reason.[2] Handling temporal streams of observations is essential for a fully-automated diagnostic system. A future version of PDE will include this extension.

7.3.3 Mechanisms of Failure

Finding a mechanism explaining how and why a failure has occurred should increase the plausibility of a given diagnostic candidate. Likewise, the inability to find a plausible failure mechanism in a well-understood domain should cast considerable doubt on a candidate model of the failed device. Failure mechanisms are part of the wealth of knowledge which a human diagnostician brings to the diagnostic task. If we are to approach human-level abilities in automated diagnosis, this important class of constraint must receive its due consideration.

It is believed that the process-centered models of QP theory are well-suited to modeling failure mechanisms. However, domain theories must be refined and extended significantly before they can provide anything close to a complete set of possible failure mechanisms. Processes such as corrosion, fracture, elastic and plastic deformation represent only a partial list. Still, as domain theories become more refined, the use of failure mechanisms in evaluating diagnostic candidates will no doubt become an important component in future diagnostic systems.

7.3.4 Active Diagnosis

Diagnosis is not done from an armchair; it often involves actively acquiring additional information about a failed system. This requires interacting with the failed system—observing, probing, even modifying it in order to understand its condition. *Active diagnosis* involves the selection of actions to be performed to a failed artifact for the purpose of obtaining useful information about the nature of the failure. Actions are taken to test or distinguish hypotheses, either by better understanding the system's current behavior, or by modifying the device in an attempt to repair it or otherwise alter its behavior. For example, pulling a spark plug wire on an automobile engine tells a mechanic a great deal about the condition of the corresponding cylinder. Generating test input patterns and selecting probes are but two of the many ways to obtain information about a failed device; active diagnosis beyond these two areas has received little attention. Researchers in theory revision [65] address the related topic of experiment planning; their work is highly relevant to diagnosis, and deserves a closer look from model-based diagnosis researchers. Future diagnostic systems will need to consider every available action if they are to approach human levels of diagnostic proficiency.

[2]A better understanding of the mechanism of the intermittency would allow us to construct a single model which includes both the correct and failed modes of behavior.

7.4 Summary

This thesis has presented *process-based diagnosis*: an approach to model-based diagnosis using the process-centered models of QP theory. Diagnosis is viewed as an attempt to repair a faulty model of a device, by minimally perturbing the structural description of the device until it yields predictions consistent with observed behavior. A portion of the model—the domain theory—is held beyond suspicion, and so is available to illuminate the search for a modified scenario description. By recognizing that the system continues to operate "correctly" according to the laws of nature, we may apply a single domain theory both to derive the model and to repair it.

This thesis echos the view of QP theory that process-centered qualitative models provide a natural way of expressing the possible behaviors within a domain. A process-centered qualitative domain theory provides the basis for both suggesting and evaluating candidate diagnoses. This research demonstrates that the assumption of a complete domain theory allows us to explain observed symptoms without relying on explicit fault models, thereby increasing robustness for novel faults.

The two-stage approach of process-based diagnosis is an effective example of the divide-and-conquer paradigm in AI. Candidates generated by the minimal retraction algorithm of Chapter 4 may contain violated closed-world assumptions, which are further explained using the abductive search algorithm of Chapter 5. This combination allows the scenario description to be modified through both additions and deletions of structural components and connections. This ability to efficiently search the space of structural descriptions provides the robustness of the process-based diagnostic approach.

This approach to diagnosis has been implemented as a LISP program, utilizing a modified ATMS (`CATMS`) and an incremental qualitative envisioner (`IQE`). The Process-based Diagnostic Engine (`PDE`) has been tested on a number of systems and fault types, providing empirical evidence that the approach is viable. Future versions of `PDE` will no doubt reduce or eliminate many of the current inefficiencies and other limitations, and continued effort in domain theory development will extend its applicability to other domains.

As systems become more complex, the need for robust, efficient diagnostic approaches can only increase. It is hoped that this research has in some small way contributed to the pool of knowledge in model-based diagnosis. As long as future researchers can find solid footing in the work of their predecessors, they will continue the climb, and the highest summits of AI will inevitably be reached.

Appendix A

Domain Theory for Fluids

;;;; Physical Objects:

```
(defEntity Physob
  (when_quantity (mass ?self) (Non-Negative (mass ?self)))
  (when_quantity (temperature ?self :ABS) (Thermal_Physob ?self))
  (when_quantity (heat ?self) (Thermal_Physob ?self))
  (when_quantity (volume ?self) (Volumetric_Physob ?self)))

(defEntity Massed_Physob
  (Physob ?self)  (Positive (mass ?self)))

(defEntity Thermal_Physob
  (Physob ?self)
  (Positive (temperature ?self :ABS))
  (Non-Negative (heat ?self))
  (:= (temperature ?self :ABS) (/ (heat ?self) (mass ?self)))
  (CONSIDER (Thermal_Properties ?self)))

(==> (and (Physob ?phob) (CONSIDER (Thermal_Properties ?phob)))
     (Thermal_Physob ?phob))

(defEntity Volumetric_Physob
  (Physob ?self)
  (Non-Negative (volume ?self))
  (when_quantity (mass ?self) (Same_Sign (mass ?self) (volume ?self)))
  (CONSIDER (Volumetric_Properties ?self)))

(==> (and (Physob ?phob) (CONSIDER (Volumetric_Properties ?phob)))
     (Volumetric_Physob ?phob))

(=TR=> (level ?X) (height ?X))
```

;;;; Containers:

```
(defEntity Container
  (Physob ?self)
  (Positive (volume ?self))
  (Non-Negative (volume (liquid_in ?self)))
  (Non-Negative (pressure (gas_in ?self) :ABS))
  (==> (DISALLOW (Overflowing ?self))
       (< (volume (liquid_in ?self)) (volume ?self)))
  (==> (CONSIDER Gravity)  (Heighted_Container ?self))
  (==> (DISALLOW (Evacuated ?self))
       (Positive (pressure (gas_in ?self) :ABS)))
  (==> (DISALLOW (Empty ?self))  (Positive (volume (liquid_in ?self))))
  (iff (Wet ?self)        (Positive (volume (liquid_in ?self))))
  (iff (Has_Gas ?self)  (Positive (pressure (gas_in ?self) :ABS)))
  (Fact_To_Close (Wet ?self))
  (Fact_To_Close (Has_Gas ?self)))

(defEntity Heighted_Container
  (> (height (top ?self)) (height (bottom ?self)))
  (M+ (height (liquid_in ?self)) (volume (liquid_in ?self)))
  (Same_Rel ((height (liquid_in ?self)) (height (top ?self)))
            ((volume (liquid_in ?self)) (volume ?self)))
  (Same_Rel ((height (liquid_in ?self)) (height (bottom ?self)))
            ((volume (liquid_in ?self)) ZERO)))

(defFact (Positive (pressure :GAGE :ABS)))

(defEntity Open_Container
  (Relative_Quantity (pressure (gas_in ?self) :GAGE))
  (Zero (pressure (gas_in ?self) :GAGE)))

(defEntity Closed_Container
  (Container ?self)
  (:= (volume (gas_in ?self)) (- (volume ?self) (volume (liquid_in ?self))))
  (Non-Negative (volume (gas_in ?self))))

(defView (Overflowing ?can)
  Individuals ((?can :type Open_Container))
  Model_Asns  ((DISTINGUISH (Can_Status ?can)))
  Conditions  ((> (volume (liquid_in ?can)) (volume ?can)))
  Relations   ((Unsafe_Condition ?can)))

(defView (Empty ?can)
  Individuals ((?can :type Container))
  Model_Asns  ((DISTINGUISH (Can_Status ?can)))
  Conditions  ((Zero (volume (liquid_in ?can)))))

(defView (Full ?can)
  Individuals ((?can :type Container))
  Model_Asns  ((DISTINGUISH (Can_Status ?can)))
  Conditions  ((= (volume (liquid_in ?can)) (volume ?can))))
```

```
(defView (Wet_By_Contained_Liquid ?cl)
  Individuals ((?cl :type Contained_Liquid :form (C_S ?sub LIQUID ?can)))
  Conditions  ((Positive (mass ?cl)))
  Relations   ((Wet ?can)))

(defView (Has_Gas ?can)
  Individuals ((?cg :type Contained_Gas :form (C_S ?sub GAS ?can)))
  Conditions  ((Positive (mass ?cg))))

(defView (Contained_Stuff (C_S ?sub ?st ?can))
  Individuals ((?can :type Container) (?sub :type Substance) (?st :type Fluid_State))
  Model_Asns  ((CONSIDER (C_S ?sub ?st ?can))))

(defView (Positive_Contained_Stuff ?cs)
  Instance_Of ((Contained_Stuff ?cs))
  Model_Asns  ((DISALLOW (Zero_Mass ?cs)))
  Relations   ((Positive (mass ?cs))))

(defEntity Contained_Stuff (Volumetric_Physob ?self))

(defEntity (Contained_Stuff (C_S ?sub LIQUID ?can)) (Contained_Liquid ?self))
(defEntity (Contained_Stuff (C_S ?sub GAS    ?can)) (Contained_Gas    ?self))

(defEntity (Contained_Liquid (C_S ?sub LIQUID ?can))
  (when (Consider (Density ?self))
    (:= (volume ?self) (/ (mass ?self) (density ?self))))
  (when (not (Consider (Density ?self)))
    (MO+ (volume ?self) (mass ?self)))
  (Sum_Member (volume (liquid_in ?can)) (volume ?self))
  (:= (height ?self) (height (liquid_in ?can))))

(defEntity (Contained_Gas (C_S ?sub gas ?c))
  (when (Consider (Density ?self))
    (:= (density ?self) (/ (mass ?self) (volume ?self))))
  (Non-Negative (pressure ?self :ABS))
  (Sum_Member (pressure (gas_in ?c) :ABS) (pressure ?self :ABS))
  (:= (volume ?self) (volume (gas_in ?c))))

(defView (defPressure_Via_Ideal_Gas_Law ?cg)
  Individuals ((?cg :type Contained_Gas :form (C_S ?sub GAS ?can)))
  Model_Asns  ((CONSIDER (thermal_properties ?cg)))
  Relations   ((:= (pressure ?cg :ABS) (/ (heat ?cg) (volume ?cg)))))

(defView (defPressure_Approx_Sans_Thermal_Properties ?cg)
  Individuals ((?cg :type Contained_Gas :form (C_S ?sub GAS ?can)))
  Model_Asns  ((not (CONSIDER (thermal_properties ?cg))))
  Relations   ((:= (pressure ?cg :ABS) (/ (mass ?cg) (volume ?cg)))))
```

```
;;;; General Flow:

(defProcess (Flow ?qty_type ?ob1 ?ob2 ?rate . ?rest)
  Instance_of ((Flow_Rel ?qty_type ?ob1 ?ob2 ?rate . ?rest))
  Influences  ((I+ (?qty_type ?ob2) ?rate)  (I- (?qty_type ?ob1) ?rate)))

(==> (not (Flow_rel ?qty_type ?ob1 ?ob2 ?rate . ?rest)) (Zero ?rate))

(defClosed_Predicate! Flow_Rel)
(defClosed_Predicate! Directed_Flow)

(defView (Positive_Flow ?qty_type ?ob1 ?ob2 ?rate)
  Instance_of ((Flow_Rel ?qty_type ?ob1 ?ob2 ?rate :ASSUME_RATE))
  Conditions  ((Positive ?rate))
  Relations   ((Directed_Flow ?qty_type ?ob1 ?ob2 ?rate)))

(defView (Negative_Flow ?qty_type ?ob1 ?ob2 ?rate)
  Instance_of ((Flow_Rel ?qty_type ?ob1 ?ob2 ?rate :ASSUME_RATE))
  Conditions  ((Negative ?rate))
  Relations   ((Directed_Flow ?qty_type ?ob2 ?ob1 (- ?rate))))

(defView (Zero_Flow ?qty_type ?ob1 ?ob2 ?rate)
  Instance_of ((Flow_Rel ?qty_type ?ob1 ?ob2 ?rate :ASSUME_RATE))
  Conditions  ((Zero ?rate)))
```

```
;;;; Fluid Flow: (free flow, pumped flow, etc.)

(defView (Supports_Flow :FWD ?rate ?src_cs)
  Instance_Of ((Exposed_to ?p1 ?src_cs))
  Individuals ((?path :conditions (Connection ?path ?p1 ?p2))
               (?p1 :form (?src_pn ?src_can))
               (?src_cs :form (C_S ?sub ?st ?src_can))
               (?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))))

(defView (Supports_Flow :BWD ?rate ?src_cs)
  Instance_Of ((Exposed_to ?p2 ?src_cs))
  Individuals ((?path :conditions (Connection ?path ?p1 ?p2))
               (?p2 :form (?src_pn ?src_can))
               (?src_cs :form (C_S ?sub ?st ?src_can))
               (?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))))

(defView (Fluid_Flow ?type ?rate ?src_cs) ;; Forward Flow;
  Instance_Of ((DefFlow_Rate ?type ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))
               (?p1 :form (?n1 ?c1))  (?p2 :form (?n2 ?c2))
               (?src_cs :form (C_S ?sub ?st ?c1)))
  Relations   ((Flow_rel MASS (C_S ?sub ?st ?c1) (C_S ?sub ?st ?c2) ?rate)))

(defView (Fluid_Flow ?type ?rate ?src_cs) ;; Backward Flow;
  Instance_Of ((DefFlow_Rate ?type ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))
               (?p1 :form (?n1 ?c1))  (?p2 :form (?n2 ?c2))
               (?src_cs :form (C_S ?sub ?st ?c2)))
  Relations   ((Flow_rel MASS (C_S ?sub ?st ?c1) (C_S ?sub ?st ?c2) ?rate)))

;;;; Thermal Effects of Fluid Flow:

(defView (Thermal_Fluid_Flow ?src_cs ?hrate ?frate)
  Instance_of ((Fluid_Flow ?type ?frate ?src_cs))
  Individuals ((?frate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))
               (?p1 :form (?n1 ?c1))  (?p2 :form (?n2 ?c2))
               (?src_cs :form (C_S ?sub ?st ?c))
               (?hrate :form (thermal_flow_rate ?path ?p1 ?p2 ?sub ?st)))
  Model_Asns  ((CONSIDER (Thermal_Flow ?path)))
  Relations   ((:= ?hrate (* ?frate (temperature ?src_cs :ABS)))
               (Flow_Rel HEAT (C_S ?sub ?st ?c1) (C_S ?sub ?st ?c2) ?hrate)))

(defView (Thermal_Fluid_Flow_PV_Work ?src_cg ?pvrate)
  Instance_of ((Thermal_Fluid_Flow ?src_cg ?hrate ?frate))
  Individuals ((?frate :form (flow_rate ?path ?p1 ?p2 ?sub GAS))
               (?p1 :form (?n1 ?c1))  (?p2 :form (?n2 ?c2))
               (?pvrate :form (PV_work_rate ?path ?p1 ?p2 ?sub)))
  Relations   ((MO ?rate ?frate)
               (Flow_Rel HEAT (C_S ?sub GAS ?c1) (C_S ?sub GAS ?c2) ?pvrate)))
```

```
;;;; Generic Fluid Flow Paths & Portals:

(defEntity Fluid_Path (Path ?self))
(defEntity Path (Physob ?self))

(defEntity (Portal (?pn ?can))
  (Fact_To_Close (Wet ?self))
  (Relative_Quantity (pressure ?self (gas_in ?can)))
  (Non-Negative (pressure ?self (gas_in ?can))))

(defView (Exposed_To ?pt ?cl)
  Instance_Of ((Wet_By_Contained_Liquid ?cl))
  Individuals ((?cl :form (C_S ?sub LIQUID ?can))
               (?pt :type Portal :form (?pn ?can)))
  Model_Asns  ((CONSIDER Gravity))
  Conditions  ((> (height (liquid_in ?can)) (height ?pt)))
  Relations   ((Wet ?pt)))

(defView (DefPressure ?pt ?cl)
  Instance_Of ((Exposed_To ?pt ?cl))
  Individuals ((?cl :form (C_S ?sub LIQUID ?can))
               (?pt :form (?pn ?can)))
  Model_Asns  ((CONSIDER (Density ?cl)))
  Relations   ((:= (rel-height (liquid_in ?can) ?pt)
   (- (height (liquid_in ?can)) (height ?pt)))
      (:= (pressure ?pt (gas_in ?can))
   (* (density ?cl) (rel-height (liquid_in ?can) ?pt)))))

(defView (DefPressure ?pt ?cl)
  Instance_Of ((Exposed_To ?pt ?cl))
  Individuals ((?cl :form (C_S ?sub LIQUID ?can))
               (?pt :form (?pn ?can)))
  Model_Asns  ((not (CONSIDER (Density ?cl))))
  Relations   ((M+ (pressure ?pt (gas_in ?can)) (height (liquid_in ?can)))
      (Positive (pressure ?pt (gas_in ?can)))))

(defView (Dry ?pt)
  Individuals ((?pt :type Portal :form (?p ?can)
                    :conditions (not (Wet ?pt))))
  Relations   ((Zero (pressure ?pt (gas_in ?can)))))

(defView (Exposed_to (?p ?can) ?cg)
  Instance_Of ((Dry (?p ?can)))
  Individuals ((?cg :type Contained_Gas :form (C_S ?sub GAS ?can)))
  Conditions  ((Positive (mass ?cg))))

(defView (zero_g_portal ?pt)
  Individuals ((?pt :type Portal :form (?p ?can)))
  Model_Asns  ((not (CONSIDER Gravity)))
  Relations   ((Zero (pressure ?pt (gas_in ?can)))))
```

;;;; Fluid Free Flow: (PIPES)

```
(=TR=> (Fluid_Connection ?path (?pn1 ?c1) (?pn2 ?c2))
       (and (Connection ?path (?pn1 ?c1) (?pn2 ?c2))
    (Container ?c1) (Container ?c2) (Pipe ?path)))

(==> (and (Connection ?path ?p1 ?p2) (Pipe ?path))
     (Pipe_Connection ?path ?p1 ?p2))

(defPredicate (Pipe_Connection ?path (?pn1 ?c1) (?pn2 ?c2))
  (portal (?pn1 ?c1))  (portal (?pn2 ?c2)))

(defPredicate (Pipe_Connection ?path ?p1 ?p2)
  (Relative_Quantity (pressure ?p1 ?p2))
  (Fluid_Path_Connection ?path ?p1 ?p2))

(defEntity Pipe  (Fluid_Path ?self))

(defView (Allows_Flow :PIPED ?rate ?src_cs)
  Instance_of ((Supports_Flow :FWD ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))
               (?path :type Pipe))
  Conditions  ((Aligned ?path)
               (> (pressure ?p1 :ABS) (pressure ?p2 :ABS))))

(defView (Allows_Flow :PIPED ?rate ?src_cs)
  Instance_of ((Supports_Flow :BWD ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st))
               (?path :type Pipe))
  Conditions  ((Aligned ?path)
               (< (pressure ?p1 :ABS) (pressure ?p2 :ABS))))

(defView (DefFlow_Rate      :PIPED ?rate ?src_cs)
  Instance_of ((Allows_Flow :PIPED ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st)))
  Model_Asns  ((not (CONSIDER (fluid_conductance ?path))))
  Relations   ((MO+ ?rate (pressure ?p1 ?p2))))

(defView (DefFlow_Rate      :PIPED ?rate ?src_cs)
  Instance_of ((Allows_Flow :PIPED ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?path ?p1 ?p2 ?sub ?st)))
  Model_Asns  ((CONSIDER (fluid_conductance ?path)))
  Relations   ((:= ?rate (* (pressure ?p1 ?p2) (fluid_conductance ?path)))
               (Non-Negative (fluid_conductance ?path))
               (if (Aligned ?path)
                   (Positive (fluid_conductance ?path))
                   (Zero     (fluid_conductance ?path)))
               (Fact_To_Close (not (Aligned ?path)))))
```

;;;;Pumped Flow:

```
(=TR=> (Pump_Connection ?pump ?p1 ?p2)
       (and (Connection ?pump ?p1 ?p2)
            (Pump ?pump)  (Portal ?p1)  (Portal ?p2)))

(==> (and (Connection ?pump ?p1 ?p2)
          (Pump ?pump)  (Portal ?p1)  (Portal ?p2))
     (%Pump_Connection ?pump ?p1 ?p2))

(defPredicate (%Pump_Connection ?pump ?pt1 ?pt2)
  (Fluid_Pump ?pump)
  (Portal ?pt1)  (Portal ?pt2)
  (Fluid_Path_Connection ?pump ?pt1 ?pt2))

(defEntity Pump (Fluid_Pump ?self))

(defEntity Fluid_Pump
  (Abstract_Fluid_Path ?self)
  (Non-Negative (conductance ?self))
  (Non-Negative (spec_flow_rate ?self)))

(defView (DefFlow_Rate  :PUMPED    ?rate ?src_cs)
  Instance_of ((Supports_Flow :FWD ?rate ?src_cs))
  Individuals ((?rate :form (flow_rate ?pump ?p1 ?p2 ?sub ?st))
               (?pump :type Fluid_Pump))
  Conditions  ((On ?pump))
  Relations   ((:= ?rate (spec_flow_rate ?pump))))

(defView (Pumped_Flow ?rate)
  Instance_of ((Fluid_Flow :PUMPED ?rate ?src_cs)))

(=TR=> (Off ?pump) (not (On ?pump)))

(defView (Pump_On ?pump)
  Individuals ((?pump :type Fluid_Pump))
  Conditions  ((On ?pump))
  Relations   ((Positive (spec_flow_rate ?pump))))

(defView (Pump_Off ?pump)
  Individuals ((?pump :type Fluid_Pump))
  Conditions  ((not (On ?pump)))
  Relations   ((Zero (spec_flow_rate ?pump))))

(defView (Variable_Pump ?pump)
  Individuals ((?pump :type Fluid_Pump
                      :conditions (Connection ?pump ?pt1 ?pt2)))
  Model_Asns ((CONSIDER (variable_pump ?pump)))
  Relations ((Pipe (internal_pipe ?pump))
             (Connection (internal_pipe ?pump) ?pt2 ?pt1)
             (Aligned (internal_pipe ?pump))
             (not (CONSIDER (fluid_conductance (internal_pipe ?pump))))))
```

;;;; Heat Flow:

```
(==> (Heat_Connection ?path ?from ?to)  (Heat_Path ?path))

(defEntity Heat_Sink
  (Thermal_Physob ?self)  (Consumer_of ?self (heat ?self)))

(defEntity Heat_Source
  (Thermal_Physob ?self)  (Producer_of ?self (heat ?self)))

(defView (Portal (?pt ?can))
  Individuals ((?hp  :conditions (Heat_Connection ?hp ?src (?pt ?can)))
               (?can :type Container)))

(defView (Portal (?pt ?can))
  Individuals ((?hp  :conditions (Heat_Connection ?hp (?pt ?can) ?dst))
               (?can :type Container)))

(defView (Heat_Connection ?hp ?src ?cs)
  Individuals ((?cs  :type Contained_Stuff :form (C_S ?sub ?st ?can)
                     :conditions (Exposed_To (?pt ?can) ?cs))
               (?hp  :conditions (Heat_Connection ?hp ?src (?pt ?can)))))

(defView (Heat_Connection ?hp ?cs ?dst)
  Individuals ((?cs  :type Contained_Stuff :form (C_S ?sub ?st ?can)
                     :conditions (Exposed_To (?pt ?can) ?cs))
               (?hp  :conditions (Heat_Connection ?hp (?pt ?can) ?dst))))

(defView (Thermal_Conduction ?path)
  Individuals ((?path :type Heat_Path))
  Model_Asns  ((CONSIDER (Thermal_Conductance ?path)))
  Relations   ((Positive (thermal_conductance ?path))))

(defView (Heat_Flow ?src ?dst ?path ?rate)
  Individuals ((?path :conditions (Heat_Connection ?path ?src ?dst))
               (?src  :type Physob) (?dst  :type Physob)
               (?rate :form (heat_flow_rate ?path ?src ?dst)))
  Conditions  ((Positive (Heat ?src)) (Positive (Heat ?dst)))
  Relations   ((Relative_Quantity (temperature ?src ?dst))
               (Flow_Rel HEAT ?src ?dst ?rate :ASSUME_RATE)))

(defView (No_Conductance_Heat_Flow ?src ?dst ?path)
  Instance_of ((Heat_Flow ?src ?dst ?path ?rate))
  Model_Asns  ((not (CONSIDER (Thermal_Conductance ?path))))
  Relations   ((MO ?rate (temperature ?src ?dst))))

(defView (Conductance_Heat_Flow ?src ?dst ?path)
  Instance_of ((Heat_Flow ?src ?dst ?path ?rate))
  Model_Asns  ((CONSIDER (thermal_conductance ?path)))
  Relations   ((:= ?rate (* (temperature ?src ?dst) (thermal_conductance ?path))))
```

;;;; Valves:

```
(defEntity Valve
  (Physob ?self)
  (Non-Negative (open_area ?self)))

(=TR=> (Closed ?valve) (not (Open ?valve)))

(defPredicate (Valve_in_Path ?valve ?path)
  (Valve ?valve)
  (==> (Closed ?valve) (not (Aligned ?path))))

(defView (Valve_in_Path ?valve ?path)
  Individuals ((?path :type Fluid_Path) (?valve :form (Valve_in ?path)))
  Model_Asns  ((CONSIDER (Valves ?path))))

(defView (Open ?valve)
  Individuals ((?valve :conditions (Valve_In_Path ?valve ?path)))
  Conditions  ((Positive (open_area ?valve))))

(defView (Changing_Conductance ?path)
  Individuals ((?valve :conditions (Valve_In_Path ?valve ?path) (Aligned ?path)))
  Model_asns  ((CONSIDER (Changing_Valves ?path)))
  Relations   ((MO (fluid_conductance ?path) (open_area ?valve))))

(defProcess (Ramped_Valve ?valve ?path ?rate)
  Individuals ((?valve :type Valve :conditions (Valve_In_Path ?valve ?path))
               (?rate :form (change_rate ?valve)))
  Model_Asns  ((CONSIDER (Ramped_Valves ?path)))
  Relations   ((Variable_Input (change_rate ?valve)))
  Influences  ((I+ (open_area ?valve) ?rate)))

(defView (Opening_Valve ?valve)
  Instance_of ((Ramped_Valve ?valve ?path ?rate))
  Conditions  ((Positive ?rate)))

(defView (Closing_Valve ?valve)
  Instance_of ((Ramped_Valve ?valve ?path ?rate))
  Conditions  ((Negative ?rate))
  Relations   ((Positive (open_area ?valve))))
```

```
;;;; Floats:

(defPredicate (Valve_Float_Switch ?flt ?valve ?status)
  (Float ?flt)  (Valve ?valve))

(defEntity (float (FLT ?name ?can))
  (> (height (FLT ?name ?can)) (bottom_height ?can))
  (< (height (FLT ?name ?can)) (top_height ?can)))

(defEntity (Valve (PATH_VALVE ?path))
  (Fluid_Path ?path)
  (when (Locked ?self) (Closed ?self))
  (when (Closed ?self) (not (Aligned ?path))))

(defEntity Fluid_Path  (fact_to_close! (not (aligned ?self))))

(defClosed_Predicate! Closed)
(defComputable_Relation Aligned)

(defView (Floating ?flt)
  Individuals ((?flt :type Float :form (FLT ?name ?can)))
  Conditions  ((> (height (c_s water liquid ?can)) (height ?flt))))

(defView (Dry_Float ?flt)
  Individuals ((?flt :type Float :form (FLT ?name ?can)))
  Conditions  ((< (height (c_s water liquid ?can)) (height ?flt))))

(defView (Non_Floating ?flt)
  Individuals ((?flt :type Float :form (FLT ?name ?can)))
  Conditions  ((<= (height (c_s water liquid ?can)) (height ?flt))))

(defView (Wet_Float ?flt)
  Individuals ((?flt :type Float :form (FLT ?name ?can)))
  Conditions  ((>= (height (c_s water liquid ?can)) (height ?flt))))

(defView (Float_Up_Valve_Closed ?flt ?valve)
  Instance_Of ((Wet_Float ?flt))
  Individuals ((?flt :conditions (Valve_Float_Switch ?flt ?valve :CLOSED)))
  Relations   ((Closed ?valve)))

(defView (Float_Down_Valve_Closed ?flt ?valve)
  Instance_Of ((Non_Floating ?flt))
  Individuals ((?flt :conditions (Valve_Float_Switch ?flt ?valve :OPEN)))
  Relations   ((Closed ?valve)))

(defView (Unlocked_Valve ?valve)
  Individuals ((?valve :type Valve))
  Conditions  ((not (Locked ?valve))))
```

;;;; Phase Changes—Boiling:

```
(defView (Liquid_Might_Boil ?cl ?can)
  Individuals ((?cl :type Contained_Liquid :form (C_S ?sub liquid ?can)))
  Model_Asns  ((CONSIDER (Boiling_in ?can)))
  Conditions  ((Positive (mass ?cl)))
  Relations   ((Positive (temperature (Boil ?cl) :ABS))
               (= (D (Volume ?cl)) (D (Mass ?cl)))))

(defView (variable_boiling_point ?cl)
  Instance_Of ((Liquid_Might_Boil ?cl ?can))
  Model_Asns  ((CONSIDER Variable_Boiling_Point))
  Relations   ((Qprop (temperature (boil ?cl) :ABS)
                      (pressure (gas_in ?can) :ABS))))

(defView (liquid_might_simple_boil ?cl)
  Instance_Of ((Liquid_Might_Boil ?cl ?can))
  Model_Asns  ((not (CONSIDER Complex_Boiling)))
  Relations   ((<= (temperature ?cl :ABS)
                   (temperature (boil ?cl) :ABS))))

(defView (Boiling_Allowed_In ?can)
  Instance_Of ((Liquid_Might_Boil ?cl ?can)
               (Directed_Flow HEAT ?src ?cl))
  Conditions  ((>= (temperature ?cl :ABS) (temperature (boil ?cl) :ABS)))
  Model_Asns  ((not (CONSIDER Complex_Boiling))))

(defView (Boiling_Allowed_In ?can)
  Instance_Of ((Liquid_Might_Boil ?cl ?can))
  Conditions  ((>= (temperature ?cl :ABS) (temperature (boil ?cl) :ABS)))
  Model_Asns  ((CONSIDER Complex_Boiling)))

(defview (Boiling ?cl ?cg ?can ?rate)
  Instance_Of ((Boiling_Allowed_in ?can))
  Individuals ((?cl :type Contained_Liquid :form (C_S ?sub liquid ?can))
               (?cg :form (C_S ?sub GAS ?can))
               (?rate :form (generation_rate GAS ?can)))
  Relations   ((Flow_Rel MASS ?cl ?cg ?rate)))

(defview (Simple_Boiling_Rate ?can ?hpath ?hrate)
  Instance_Of ((Boiling ?cl ?cg ?can ?brate)
               (Heat_Flow ?src ?cl ?hpath ?hrate))
  Model_Asns  ((not (CONSIDER Complex_Boiling)))
  Relations   ((Qprop0 ?brate ?hrate)
               (Qprop (temperature ?cl :ABS) (temperature (boil ?cl) :ABS))))

(defView (complex_boiling_rate ?can)
  Instance_Of ((Boiling ?cl ?cg ?can ?brate))
  Model_Asns  ((CONSIDER Complex_Boiling))
  Relations   ((Relative_Quantity (temperature ?cl (boil ?cl)))
               (:= ?brate (*0+ (temperature ?cl (boil ?cl)) (mass ?cl)))))
```

```
(defView (boiling_heat_flow ?can ?rate)
  Instance_Of ((Boiling ?cl ?cg ?can ?brate))
  Individuals ((?rate :form (heat_flow_rate (boiling ?can))))
  Model_Asns  ((CONSIDER thermal_boiling))
  Relations   ((:= ?rate (*0+ ?brate (temperature ?cl :ABS)))
               (Flow_Rel HEAT ?cl ?cg ?rate)))

(defView (Boiling_Latent_Heat_Flow ?can ?rate ?brate)
  Instance_Of ((Boiling ?cl ?cg ?can ?brate))
  Individuals ((?rate :form (latent_heat_flow_rate (boiling ?can))))
  Model_Asns  ((CONSIDER latent_heat_of_vaporization))
  Relations   ((Flow_Rel HEAT ?cl ?cg ?rate)))

(defView (simple_boiling_latent_heat_flow ?can ?lrate)
  Instance_Of ((Simple_Boiling_Rate ?can ?hpath ?hrate)
               (Boiling_Latent_Heat_Flow ?can ?lrate ?brate))
  Model_Asns  ((not (CONSIDER Complex_Boiling)))
  Relations   ((:= ?lrate ?hrate)))

(defView (complex_boiling_latent_heat_flow ?can ?lrate)
  Instance_Of ((Boiling_Latent_Heat_Flow ?can ?lrate ?brate))
  Model_Asns  ((CONSIDER Complex_Boiling))
  Relations   ((qprop0 ?lrate ?brate)))

;;;; Condensation:

(defView (Gas_Might_Condense ?cg ?can)
  Individuals ((?cg :type contained_gas :form (C_S ?sub gas ?can)))
  Model_Asns  ((CONSIDER (Condensing_in ?can)))
  Conditions  ((Positive (mass ?cg)))
  Relations   ((Positive (temperature (condense ?cg) :ABS))))

(defView (variable_boiling_point ?cg)
  Instance_Of ((Gas_Might_Condense ?cg ?can))
  Model_Asns  ((CONSIDER Variable_Boiling_Point))
  Relations   ((Qprop (temperature (condense ?cg) :ABS)
                      (pressure ?cg :ABS))))

(defView (gas_might_simple_condense ?cg)
  Instance_Of ((Gas_Might_Condense ?cg ?can))
  Model_Asns  ((not (CONSIDER Complex_Boiling)))
  Relations   ((>= (temperature ?cg :ABS)
                   (temperature (condense ?cg) :ABS))))
```

```
(defView (Condensing_Allowed_In ?can)
  Instance_Of ((Gas_Might_Condense ?cg ?can)
               (Directed_Flow HEAT ?cg ?src))
  Individuals ((?cg :form (C_S ?sub gas ?can)))
  Model_Asns  ((not (CONSIDER Complex_Boiling)))
  Conditions  ((<= (temperature ?cg :ABS) (temperature (condense ?cg) :ABS))))

(defView (Condensing_Allowed_In ?can)
  Instance_Of ((Gas_Might_Condense ?cg ?can))
  Conditions  ((<= (temperature ?cg :ABS) (temperature (condense ?cg) :ABS)))
  Model_Asns  ((CONSIDER Complex_Boiling)))

(defView (Condensing ?cg ?cl ?can ?rate)
  Instance_Of ((Condensing_Allowed_in ?can))
  Individuals ((?cg :type Contained_Gas :form (C_S ?sub GAS ?can))
               (?cl :form (C_S ?sub LIQUID ?can))
               (?rate :form (generation_rate LIQUID ?can)))
  Relations   ((Flow_Rel MASS ?cg ?cl ?rate)))

(defview (Simple_Condensing_Rate ?can)
  Instance_Of ((Condensing ?cg ?cl ?can ?crate)
               (Heat_Flow ?cg ?dst ?hpath ?hrate))
  Model_Asns  ((not (CONSIDER Complex_Boiling)))
  Relations   ((Qprop0 ?crate ?hrate)
               (Qprop (temperature ?cg :ABS)
                      (temperature (condense ?cg) :ABS))))

(defView (Complex_Condensing_Rate ?can)
  Instance_Of ((Condensing ?cg ?cl ?can ?crate))
  Model_Asns  ((CONSIDER Complex_Boiling))
  Relations   ((Relative_Quantity (temperature (condense ?cg) ?cg))
               (:= ?crate (*0+ (temperature (condense ?cg) ?cg) (mass ?cg)))))

(defView (Condensing_Heat_Flow ?can)
  Instance_Of ((Condensing ?cg ?cl ?can ?crate))
  Individuals ((?rate :form (Heat_Flow_Rate (condensing ?can))))
  Model_Asns  ((CONSIDER thermal_boiling))
  Relations   ((:= ?rate (*0+ ?crate (temperature ?cg :ABS)))
               (Flow_Rel HEAT ?cg ?cl ?rate)))
```

;;;; Miscillaneous—Ideal Sources:

```
(defProcess (Replenish ?thing ?quantity)
  Individuals ((?thing :conditions (Ideal_Source_of ?thing ?quantity)))
  Relations ((Zero (D ?quantity))
             (I+ ?quantity (replenish_rate ?thing ?quantity))
             (Positive (replenish_rate ?thing ?quantity))))

(defProcess (Positive_Source ?thing ?quantity)
  Individuals ((?thing :conditions (Producer_of ?thing ?quantity)))
  Relations ((Positive (positive_source_rate ?thing ?quantity))
             (I+ ?quantity (positive_source_rate ?thing ?quantity))))

(defProcess (Positive_Sink ?thing ?quantity)
  Individuals ((?thing :conditions (Consumer_of ?thing ?quantity)))
  Relations ((Positive (positive_sink_rate ?thing ?quantity))
             (I- ?quantity (positive_sink_rate ?thing ?quantity))))
```

;;;; Relative Quantities:

```
(=TR=> (Relative_Quantity (?qty_type ?X ?Y))
       (:= (?qty_type ?X ?Y)
           (- (?qty_type ?X :ABS) (?qty_type ?Y :ABS))))
```

;;;; Floating Inputs:

```
(defView (Vary_Variable_Input ?qty)
  Individuals ((?qty :type Variable_Input :conditions (Quantity ?qty)))
  Relations   ((Positive (Dummy_Rate :SYSTEM))
               (Qprop+ ?qty (Dummy_Ramp :SYSTEM))
               (Qprop- ?qty (Dummy_Ramp :SYSTEM))
               (Drel (Dummy_Ramp :SYSTEM) (Dummy_Rate :SYSTEM))))
```

```
;;;; Tolerances:

(defView (Bounded ?qty)
  Individuals ((?qty :type Quantity :conditions (Nominal_Value ?qty ?nom)
                       (Tolerance ?qty ?tol)))
  Model_Asns  ((CONSIDER (Tolerances ?qty)))
  Relations   ((:= (lower_limit ?qty) (- ?nom ?tol))
               (:= (upper_limit ?qty) (+ ?nom ?tol))
               (Positive ?tol)))

(defView (Under_Tolerance ?qty)
  Individuals ((?qty :type Quantity :conditions (Quantity (lower_limit ?qty))))
  Conditions  ((< ?qty (lower_limit ?qty)))
  Model_Asns  ((CONSIDER (Tolerances ?qty)))
  Relations   ((Unsafe_Condition ?qty)))

(defview (Over_Tolerance ?qty)
  Individuals ((?qty :type Quantity :conditions (Quantity (upper_limit ?qty))))
  Conditions  ((> ?qty (upper_limit ?qty)))
  Model_Asns  ((CONSIDER (Tolerances ?qty)))
  Relations   ((Unsafe_Condition ?qty)))

;;;; Steady state:

(==> (Steady_State_Quantity ?Q) (Constant ?Q))

(==> (and (CONSIDER Steady_State) (Quantity ?Qty))
     (Steady_State_Quantity ?qty))

(==> (and (CONSIDER (Steady_State_Individual ?ind))
          (Quantity (?qty_type ?ind . ?rest)))
     (Steady_State_Quantity (?qty_type ?ind . ?rest)))

(==> (and (CONSIDER (Steady_State_Quantity_Type ?qty_type))
          (Quantity (?qty_type . ?inds)))
     (Steady_State_Quantity (?qty_type . ?inds)))
```

;;;;Modeling Assumptions:

```
(=TR=> (Wildcard ?x)  (:TEST (eq ?x '*)))

(defFact (State SOLID))
(defFact (Fluid_State LIQUID))
(defFact (Fluid_State GAS))

(==> (and (Consider (C_S ?sub ?st *))
  (Substance ?sub) (State ?st) (Container ?can))
     (Consider (C_S ?sub ?st ?can)))

(==> (and (Consider (C_S ?sub * ?can))
  (Substance ?sub) (Consider (State ?st))
  (or (Wildcard ?can) (Container ?can)))
     (Consider (C_S ?sub ?st ?can)))

(==> (and (Consider (C_S * ?st ?can)) (Substance ?sub)
  (or (Wildcard ?st)  (State ?st))
  (or (Wildcard ?can) (Container ?can)))
     (Consider (C_S ?sub ?st ?can)))

(==> (Consider (Substance ?sub))  (Substance ?sub))

(==> (and (Disallow (Zero_Mass OB*))  (Physob ?ob))
     (Disallow (Zero_Mass ?ob)))

(==> (and (Distinguish (Can_Status CAN*))  (Container ?can))
     (Distinguish (Can_Status ?can))

(==> (and (Consider (Boiling_In CAN*))  (Container ?can))
     (Consider (Boiling_In ?can)))

(==> (and (Consider (Condensing_In CAN*))  (Container ?can))
     (Consider (Condensing_In ?can)))

(==> (and (Consider (Level_Path PIPE*))  (Pipe ?pipe))
     (Consider (Level_Path ?pipe)))

(==> (and (Consider (Level_Path ?pipe))
  (Connection ?pipe ?pt1 ?pt2))
     (= (height ?pt1) (height ?pt2)))

(==> (and (Consider (Fluid_Conductance PIPE*))  (Pipe ?pipe))
     (Consider (Fluid_Conductance ?pipe)))

(==> (and (Consider (Aligned PIPE*))  (Pipe ?pipe))
     (Aligned ?pipe))

(==> (and (Consider (On PUMP*))  (Pump ?pump))
     (On ?pump))
```

References

[1] Danny Bobrow, editor. *Qualitative Reasoning about Physical Systems.* MIT Press, Cambridge, MA, 1985.

[2] W. J. Clancey. The epistemology of a rule-based expert system—a framework for explanation. *Artificial Intelligence*, 20:215–251, 1983.

[3] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[4] John W. Collins. Ontological and algorithmic explorations in qualitative reasoning. Master's thesis, University of Illinois at Urbana-Champaign, May 1988.

[5] John W. Collins and Dennis DeCoste. CATMS: An ATMS which avoids label explosions. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, July 1991.

[6] John W. Collins and Kenneth D. Forbus. Reasoning about fluids via molecular collections. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, 1987.

[7] John W. Collins and Kenneth D. Forbus. Building qualitative models of thermodynamic processes. Technical report, University of Illinois at Urbana-Champaign, 1993. (In preparation).

[8] Luca Console and Pietro Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, 7(3):133–141, 1991.

[9] James Crawford, Adam Farquhar, and Benjamin Kuipers. QPC: A compiler from physical models into qualitative differential equations. In *Proceedings of the Nineth National Conference on Artificial Intelligence*, pages 365–372, July 1990.

[10] Judith Crow and John Rushby. Model-based reconfiguration: Toward an integration with diagnosis. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 836–841, 1991.

[11] Bruce D'Ambrosio and James Edwards. A partitioned ATMS. In *Proceedings of the Seventh IEEE Conference on AI Applications*, pages 330–336, 1991.

[12] Randall Davis. Diagnosis based on structure and behavior. In *Proceedings of the Second National Conference on Artificial Intelligence*, 1982.

[13] Randall Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24, 1984.

[14] Randall Davis and Walter C. Hamscher. *Model-Based Reasoning: Troubleshooting*, pages 297–346. Morgan Kaufmann, San Mateo, CA., 1988.

[15] Johan de Kleer. How circuits work. In D. Bobrow, editor, *Qualitative Reasoning about Physical Systems.* The MIT Press, 1984.

[16] Johan de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28(2):127–162, March 1986.

[17] Johan de Kleer. Extending the ATMS. *Artificial Intelligence*, 28(2), March 1986.

[18] Johan de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28(2), March 1986.

[19] Johan de Kleer. Using crude probability estimates to guide diagnosis. *Artificial Intelligence*, 45:381–391, 1990.

[20] Johan de Kleer. Focusing on probable diagnoses. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 842–848, 1991.

[21] Johan de Kleer and Daniel G. Bobrow. Qualitative reasoning with higher-order derivatives. In *Proceedings of the Fourth National Conference on Artificial Intelligence*, pages 86–91, August 1984.

[22] Johan de Kleer and John Seeley Brown. Model-based diagnosis in SOPHIE III. In *Readings in Model Based Diagnosis*, pages 179–205. Morgan Kaufmann, 1992.

[23] Johan de Kleer and John Seely Brown. A qualitative physics based on confluences. *Artificial Intelligence*, 24:7–83, 1984.

[24] Johan de Kleer, Alan K. Mackworth, and Raymond Reiter. Characterizing diagnoses. In *Proceedings of the Nineth National Conference on Artificial Intelligence*, pages 324–330, 1990.

[25] Johan de Kleer and Brian Williams. Reasoning about multiple faults. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 132–139, August 1986.

[26] Johan de Kleer and Brian Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32, 1987.

[27] Johan de Kleer and Brian Williams. Diagnosis with behavioral modes. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1324–1330, 1989.

[28] Dennis DeCoste. Dynamic across-time measurement interpretation: Maintaining qualitative understandings of physical system behavior. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, October 1989. (Technical Report UIUCDCS-R-90-1572, University of Illinois at Urbana-Champaign, February 1990).

[29] Dennis DeCoste and John Collins. CATMS: An ATMS which avoids label explosions. Technical report, Institute for the Learning Sciences, Northwestern University, 1991.

[30] Dennis DeCoste and John Collins. IQE: An incremental qualitative envisioner. In *Proceedings of the Fifth Workshop on Qualitative Physics*, December 1991.

[31] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12:231–272, 1979.

[32] David W. Etherington, Kenneth D. Forbus, Matthew L. Ginsberg, David Israel, and Vladimir Lifschitz. Critical issues in nonmonotonic reasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 500–504, 1989.

[33] Brian Falkenhainer. *Learning from Physical Analogies: A Study in Analogy and the Explanation Process.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, December 1988. (Technical Report UIUCDCS-R-88-1479, University of Illinois at Urbana-Champaign, December 1988).

[34] Brian Falkenhainer and Kenneth D. Forbus. Setting up large-scale qualitative models. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 301–306, August 1988.

[35] Brian Falkenhainer and Shankar Rajamoney. The interdependencies of theory formation, revision, and experimentation. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 353–366, Ann Arbor, MI, June 1988.

[36] Nicholas S. Flann, Thomas G. Dietterich, and Dan R. Corpron. Forward chaining logic programming with the ATMS. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 24–29, 1987.

[37] Kenneth D. Forbus. Qualitative process theory. *Artificial Intelligence*, 24:85–168, 1984.

[38] Kenneth D. Forbus. Qualitative process theory. Technical Report TR-789, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA., 1984.

[39] Kenneth D. Forbus. Interpreting measurements of physical systems. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 113–117, August 1986. (Technical Report UIUCDCS-R-86-1248, University of Illinois at Urbana-Champaign, March 1986).

[40] Kenneth D. Forbus. The qualitative process engine. Technical Report UIUCDCS-R-86-1288, University of Illinois at Urbana-Champaign, December 1986.

[41] Kenneth D. Forbus. The qualitative process engine. In Daniel S. Weld and Johan de Kleer, editors, *Readings in Qualitative Reasoning about Physical Systems*, pages 220–235. Morgan Kaufmann, 1990. (Technical Report UIUCDCS-R-86-1288, University of Illinois at Urbana-Champaign, December 1986).

[42] Kenneth D. Forbus and Johan de Kleer. Focusing the ATMS. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 193–198, August 1988.

[43] Kenneth D. Forbus and Johan de Kleer. *Building Problem Solvers.* MIT Press, Cambridge, MA., 1993.

[44] Gerhard Friedrich, Georg Gottlob, and Wolfgang Nejdl. Physical impossibility instead of fault models. In *Proceedings of the Nineth National Conference on Artificial Intelligence*, pages 331–336, 1990.

[45] Michael R. Genesereth. Diagnosis using hierarchical design models. In *Proceedings of the Second National Conference on Artificial Intelligence*, 1982.

[46] Michael R. Genesereth. The use of design descriptions in automated diagnosis. *Artificial Intelligence*, 24:411–437, 1984.

[47] Walter Hamscher. Temporally coarse representation of behavioir for model-based troubleshooting of digital circuits. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 888–893, 1989.

[48] Patrick J. Hayes. The naive physics manifesto. In Donald Michie, editor, *Expert Systems in the Micro-Electronic Age.* Edinburgh University Press, 1979.

[49] Patrick J. Hayes. Naive physics 1: Ontology for liquids. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World.* Ablex, 1985.

[50] Patrick J. Hayes. The second naive physics manifesto. In J. Hobbs and R. Moore, editors, *Formal Theories of the Commonsense World.* Ablex, 1985.

[51] J. Hobbs and R. Moore, editors. *Formal Theories of the Commonsense World.* Ablex, 1985.

[52] Caroline N. Koff, Nicholas S. Flann, and Thomas G. Dietterich. An ATMS for equivalence relations. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 182–187, 1988.

[53] Ben Kuipers. Commonsense reasoning about causality: Deriving behavior from structure. *Artificial Intelligence*, 24:169–203, 1984.

[54] Ben Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:289–338, 1986.

[55] John McCarthy and Pat Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, volume 4, pages 463–502. Edinburgh University Press, 1969.

[56] Hwee Tou Ng and Raymond J. Mooney. On the role of coherence in abductive explanation. In *Proceedings of the Nineth National Conference on Artificial Intelligence*, pages 337–342, 1990.

[57] Hwee Tou Ng and Raymond J. Mooney. An efficient first-order abduction system based on the ATMS. Technical Report AI 91-151, Artificial Intelligence Lab, University of Texas at Austin, 1991.

[58] David Poole. Normality and faults in logic-based diagnosis. In *IJCAI-89*, pages 1304–1310, 1989.

[59] O. Raiman. Order of magnitude reasoning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 100–104, August 1987.

[60] Olivier Raiman, Johan de Kleer, Vijay Saraswat, and Mark Shirley. Characterizing non-intermittent faults. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 849–854, 1991.

[61] Shankar Rajamoney. Experimentation-based theory revision. In *Proceedings of the 1988 AAAI Spring Symposium Series: EBL*, Stanford, CA, March 1988.

[62] Shankar Rajamoney and Gerald DeJong. Active explanation reduction: An approach to the multiple explanations problem. In *Proceedings of the Fifth International Machine Learning Conference*, June 1988. (Technical Report UILU-ENG-88-2218).

[63] Shankar Rajamoney, Gerald DeJong, and Boi Faltings. Towards a model of conceptual knowledge acquisition through directed experimentation. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, August 1985.

[64] Shankar A. Rajamoney. Automated design of experiments for refining theories. Master's thesis, University of Illinois at Urbana-Champaign, 1986. (CSL Technical Report T-2213).

[65] Shankar A. Rajamoney. *Explanation-Based Theory Revision: An Approach to the Problems of Incomplete and Incorrect Theories.* PhD thesis, University of Illinois at Urbana-Champaign, December 1988.

[66] Raymond Reiter. On closed world data bases. In H. Gallaire and J. Minker, editors, *Logic and Data Bases.* Plenum Press, New York, 1978.

[67] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[68] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[69] Raymond Reiter and Johan de Kleer. Foundations of assumption-based truth maintenance systems: Preliminary report. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 183–188, July 1987.

[70] Ron Rymon. Search through systematic set enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.

[71] Bart Selman and Hecter J. Levesque. Abductive and default reasoning: A computational core. In *Proceedings of the Nineth National Conference on Artificial Intelligence*, pages 343–348, 1990.

[72] Yoav Shoham. Chronological ignorance: Experiments in nonmonotonic temporal reasoning. *Artificial Intelligence*, 36:279–331, 1988.

[73] E. H. Shortliffe. *Computer-Based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.

[74] Mark E. Stickel. A prolog-like inference system for computing minimum-cost abductive explanations in natural-language interpretation. Technical report, SRI International, September 1988. Technical Note 451.

[75] Peter Struss and Oskar Dressler. "Physical Negation" – Integrating fault models into the General Diagnostic Engine. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1318–1323, 1989.

[76] David Rutherford Throop. *Model-Based Diagnosis of Complex, Continuous Mechanisms.* PhD thesis, University of Texas at Austin, August 1991.

[77] Dan Weld. Comparative analysis. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy, August 1987.

[78] Dan Weld. *Theories of Comparative Analysis.* PhD thesis, Massachusetts Institute of Technology, May 1988.

[79] Brian Williams. Qualitative analysis of MOS circuits. Master's thesis, Massachusetts Institute of Technology, July 1984. (MIT AI Lab Technical Report 767).

Vita

John William Collins was born on January 16, 1957 in Fort Wayne, Indiana. He received a BS degree in Mechanical Engineering from Rose Hulman Institute of Technology in 1979. From 1979 to 1985, he worked in Product Development and Applied Research for Motorola, Inc. in Plantation, Florida.

In 1985 he enrolled as a graduate student at the University of Illinois, where from 1986 to 1992 he worked as a Research Assistant in the Qualitative Reasoning Group, under the direction of Dr. Ken Forbus. In 1988 he received his MS degree in Computer Science from the University of Illinois. Since August 1992 he has been an instructor and later an assistant professor in the Department of Electrical and Computer Engineering at the University of Miami. He received his PhD in Computer Science from the University of Illinois in 1993.

His research interests in artificial intelligence include automated diagnosis, qualitative and model-based reasoning, connectionist architectures and object-oriented programming. He is a participant in the University of Miami Medical School's Intelligent Information Technology Center, whose charter is to develop an automated filmless environment for medical imaging; and is also involved in research for the U.S. Coast Guard to develop decision support systems for oil spill countermeasures.