AN OPERATOR-BASED APPROACH TO FIRST-ORDER THEORY REVISION

BRADLEY LANCE RICHARDS

AUGUST 1992 AI 92-181

AN OPERATOR-BASED APPROACH TO FIRST-ORDER THEORY REVISION

by

BRADLEY LANCE RICHARDS, B.S.E.E, M.S.E.E, M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN
August, 1992

AN OPERATOR-BASED APPROACH TO FIRST-ORDER THEORY REVISION

Publication No. ______________

Bradley Lance Richards, Ph.D.
The University of Texas at Austin, 1992

Supervisor: Raymond J. Mooney

Knowledge acquisition is a difficult and time-consuming task, and as error-prone as any human activity. Thus, knowledge bases must be maintained, as errors and omissions are discovered. To address this task, recent learning systems have combined inductive and explanation-based techniques to produce a new class of systems performing *theory revision*. When errors are discovered in a knowledge base, theory revision allows automatic self-repair, eliminating the need to recall the knowledge engineer and domain expert.

To date, theory revision systems have been limited to propositional domains. This thesis presents a system, FORTE (First-Order Revision of Theories from Examples), that performs theory revision in first-order domains. Moving to a first-order representation creates many new challenges, such as argument selection and recursion. But it also opens many new application areas, such as logic programming and qualitative modelling, that are beyond the reach of propositional systems.

Chapter 1
INTRODUCTION

The past few years have seen a merger of inductive and explanation-based learning algorithms into a new class of systems performing theory revision. The premise of theory revision is that we can obtain a domain theory, be it from a book or an expert, but we cannot expect that theory to be entirely complete or correct. Extracting domain knowledge is a time-intensive process [Feigenbaum, 1977], and as error prone as any human activity. When errors are discovered in a rule base, theory revision allows automatic self-repair, obviating the need to recall the knowledge engineer and domain expert.

Theory revision uses pre-classified training data to improve a theory. Once a theory is known to be faulty, the user gathers examples of the data that the theory fails on, together with past instances that the theory should continue to work on, and passes this data and the faulty theory to a theory revision system. The system specializes or generalizes various rules, or creates new ones, while preserving as much of the original structure of the theory as possible. In the extreme case, where the initial theory is empty, a theory revision system can perform pure inductive learning.

To date, theory revision systems have been limited to propositional domains.[1] A propositional representation restricts them to revising theories that perform classification tasks, such as medical diagnosis or character recognition. This thesis presents a system, FORTE (First-Order Revision of

[1]This includes neural networks, which perform theory revision on non-symbolic theories represented numerically as weights and thresholds.

Theories from Examples), that performs theory revision in first-order domains. Using a first-order representation opens many new problem areas. For example, FORTE can revise simple logic programs, opening the way to automatic program debugging. Another first-order domain is that of qualitative modelling; FORTE can induce, revise, and diagnose qualitative models. FORTE can also induce and revise DCG grammars. All of these tasks are beyond the reach of a propositional system.

Devising a system to work in first-order logic presents new challenges as well. The space of possible theories in a propositional domain is finite, whereas a first-order theory space is usually infinite. A first-order system must consider arguments in theory literals, which brings with it the problems of unification. Also, first-order theories may be recursive; something that rarely arises in propositional domains.[2]

FORTE meets all of these challenges. In order to overcome the problems of an infinite search space, FORTE uses a hill-climbing algorithm; the heuristics that guide this hill-climbing are effective enough that FORTE does not use backtracking. Unification is handled through a Prolog-like meta-interpreter. Lastly, recursive theories are fully supported, although they require special training sets.

Another challenge to a first-order system is the wide variety of domains to which it may be applied. Wray Buntine, in [Buntine, 1990], points out that

[2]Recurrent neural nets are an example of a propositional system that uses recursion.

> There exist universal learning algorithms (and each of us provides living proof), but these can always be outperformed by a second class of algorithms better selected and modified for the particular application.

He goes on to quote [Waterman, 1986] on the "strong knowledge principle":

> ...to make a program intelligent, provide it with lots of high quality specific knowledge about some problem area.

One of the objectives of this research has been to discover a set of theory revision operators that are general purpose, in that they can be used to revise theories across a wide range of first-order domains. In addition, FORTE allows the user to introduce domain-specific knowledge into the theory revision process. Thus, we hope to have the best of both worlds: a general purpose theory revision system that can make use of domain-specific knowledge to improve its performance in particular problem areas.

FORTE allows the introduction of three types of domain-specific knowledge: language bias, revision verification, and a fundamental domain theory. Language bias allows the user to force FORTE to develop certain types of theories by limiting the types of rules that may appear in the theory. Revision verification provides a place for domain-dependent consistency checks. For example, in qualitative modelling the revision verifier ensures that the dimensions of variables are consistent.[3] Lastly, the fundamental domain theory can be used to provide definitions of the basic attributes and relations in the target domain. For instance, in qualitative modelling the fundamental domain theory defines the constraints used to construct qualitative models.

[3]E.g., one would never add a number representing velocity to a number representing mass.

FORTE's primary contribution to the field of machine learning is that it is the first theory revision system to operate in first-order logic. In addition, it breaks new ground in the demonstration domains of logic programming and qualitative modelling. In logic programming, FORTE provides a unique capability to automatically revise simple programs. This holds promise, for example, as an instructional aid for novice programmers. In qualitative modelling, FORTE can induce and revise qualitative models from behavioral information, even when that information is incomplete.

The next chapter provides a summary of previous work in machine learning and in the two domains of logic programming and qualitative modelling. Chapter 3 describes our objective in developing FORTE and presents the formal model on which it is based. Chapter 4 gives an overview of FORTE's architecture and operation, and Chapter 5 discusses the FORTE algorithms in detail. Chapters 6 through 9 present empirical results. Chapter 10 compares FORTE to the most closely related work in machine learning, logic programming, and qualitative modelling. Chapter 11 gives our recommendations for future work, and Chapter 12 concludes the thesis. The appendices include training data, initial theories, and sample runs for all of the test domains discussed in Chapters 6 through 9.

Chapter 2
BACKGROUND

This chapter provides background information that may be helpful in understanding the remainder of the thesis. A discussion of related work, in which Forte will be compared to closely related work done by other researchers, is left until Chapter 10.

FORTE's development is an outgrowth of related work in propositional theory revision, first-order inductive learning, and inverse resolution. Developing a first-order theory revision system would not have been possible without prior ground-breaking work in these areas, and this previous work is discussed in the first three sections of this chapter. In addition, FORTE represents a contribution to the fields of logic program synthesis and qualitative model induction. Hence, the last two sections of this chapter discuss previous work in these two fields.

2.1 PROPOSITIONAL THEORY REVISION

A number of researchers have developed propositional theory revision systems. [Ginsberg, 1990] presents a method of theory revision that involves "reducing" a theory into a special form suitable for use by an inductive learner, performing induction, and then "retranslating" the result back into the original theory language. In [Towell, Shavlik, and Noordewier, 1990], KBANN translates the initial theory into a neural network, and revises the network using standard neural network techniques. However, extracting a revised theory from the trained network is difficult, and the results presented in [Towell, Shavlik, and Craven, 1991] are only partially satisfactory. Both of these represent hybrid approaches, which perform theory revision indirectly, by translating a theory into a new form for learning, and then extracting a revised theory from the result. Much of the structural information contained in the original theory is inevitably lost in the translation process.

KRUST, described in [Craw and Sleeman, 1991], takes the approach of generating a wide array of possible revisions to a knowledge base, and then filtering and ranking the revisions to choose the most suitable one. Much of the filtering depends on the existence of certain canonical "chest-nut"examples, which must be identified by the human expert.

EITHER, described in [Ourston and Mooney, 1990], performs direct theory revision. It identifies points in the theory that cause errors on the training set; and uses abduction to determine what theory modifications would correct each error. It uses greedy search to identify a small set of modifications that repair all the errors and revises the theory accordingly. Although EITHER is limited to propositional domains, it is the conceptual predecessor to FORTE.

2.2 FIRST-ORDER INDUCTION

Induction is the process of generating a theory purely from examination of the training set. In a typical machine learning task, each instance in the training set is defined by a list of facts and a classification. In medical diagnosis, for example, the facts would be a list of symptoms and test results, and the classification would be the disease indicated by this information. An induction system creates rules that specify what facts must be present for each classification.

One of the standard first-order induction systems is FOIL, described in [Quinlan, 1990]. FOIL learns binary concepts (i.e., instances are either "positive," meaning they are members of the concept set, or they are "negative," meaning they are not members of the set). It works by generalization, beginning with an empty theory (representing an empty concept set), and constructing a set of Horn clauses that cover the positive instances while excluding the negative ones. Each clause is constructed one literal at a time, choosing at each step the antecedent that best discriminates between the positive and negative instances.

This hill-climbing technique is efficient, but vulnerable to local maxima. In order to reduce this problem, [Quinlan, 1991] adds *determinate literals* to FOIL. A determinate literal is a literal that has only one possible binding. For example, suppose we wish to construct a merge-sort predicate from the base rule

```
merge_sort(A, B) :- true.
```

Among the literals we can add to the clause is split(A, C, D). This is a determinate literal since, for any list A, it produces unique lists C and D. FOIL adds all such determinate literals to the clause before beginning the normal induction process. This is a recursive process, as the new variables introduced by determinate literals can be used to define further determinate literals; hence, an arbitrary depth-bound is imposed. Excess determinate literals are deleted after learning is complete.

Although FORTE is a theory revision system, one of its techniques for building new rules and specializing existing ones is very similar to the basic FOIL algorithm.

2.3 INVERSE RESOLUTION

Inverse resolution is a generalization technique that can be used to perform induction. The first system based on this technique was Duce

([Muggleton, 1987]). Duce is a propositional system that takes advantage of the ease with which resolution steps can be reversed in propositional logic. For example, suppose we have the resolution step:

$$\begin{array}{ll} \leftarrow \text{alpha, beta} & \text{(goal)} \\ \text{alpha} \leftarrow \text{delta} & \text{(input clause)} \\ \hline \leftarrow \text{delta, beta} & \text{(resolvent)} \end{array}$$

If we know the resolvent and either the goal or the input clause, we can abduce the missing element. Duce uses an oracle to verify its operations.

From Duce it was a short but important step to CIGOL, a similar system working in first-order logic ([Muggleton and Buntine, 1988]). However, CIGOL has substantial limitations. For example, it assumes all input clauses are unit clauses[4], and, like Duce, it depends on an oracle. The next step in inverse resolution systems was GOLEM, described in [Muggleton and Feng, 1990]. GOLEM is a very capable induction system that learns first-order theories "bottom-up," using inverse resolution techniques to generalize the positive training instances while excluding the negative instances. However, it still requires input clauses to be unit clauses.

Two of FORTE's theory revision operators are based on inverse resolution. However, unlike CIGOL and GOLEM, FORTE's operators work on arbitrary clauses.

[4]A *unit clause* is a clause with no antecedents. CIGOL and GOLEM require all background knowledge to be pre-evaluated into an extensional form consisting only of ground facts.

2.4 LOGIC PROGRAM SYNTHESIS

Historically, there have been two major approaches to logic program synthesis. The first approach is to derive a program from a formal specification. Given a correct, possibly non-executable specification of a problem, we can transform the specification into a logic program via sound transformation rules [Lloyd and Torpor, 1984] [Tamaki and Sato, 1984], via a proof of the specification [Bundy, Smaill, and Wiggins, 1990], or via planning verification proofs [Kraan, Basin, and Bundy, 1992].

The second approach is to develop a program from a set of input-output examples. This is the approach used in [Shapiro, 1983] for automatic program debugging. However, Shapiro's approach is intended primarily as a programmer's aid, and depends heavily on the user as an oracle.

[Flener and Deville, 1991] attempts to blend these two approaches by using positive input-output examples along with an informal specification composed of "properties." A property is a nonrecursive clause that is a generalization of one or more given examples. Properties can be viewed either as nonground examples or as an initial theory. Flener further abets his program synthesis method by using program schemata and by allowing questions to an oracle. Unfortunately, while Flener's ideas appear promising, he provides no results or other means of evaluating their effectiveness.

FORTE can induce simple logic programs from a training set of input-output tuples, without recourse to an oracle. Results of testing FORTE in the domain of logic programming appear in Chapter 7.

Since FORTE is a theory revision system, it can also revise incorrect logic programs. Possibly the most well-known work in logic program debugging is PDS6 in [Shapiro, 1983]. PDS6 is intended as a programmer's aid, and is therefore highly interactive. It traces the execution of a program, and queries the user to determine which clauses in the program are incorrect and how they ought to be revised. Another approach is taken by [Murray, 1986]. Murray's work is in automated tutoring, and he uses a known-correct program in an interactive system to help a novice programmer debug an incorrect program. FORTE's application in this area is quite different from either of these approaches, since FORTE is a fully automatic system.

2.5 QUALITATIVE MODEL BUILDING

Qualitative modelling attempts to produce models that explain system behaviors in intuitive terms. For example, when trying to understand the effect of heating a pot of water, it may be more useful to simply know that the pot may boil over rather than to understand the numerical thermodynamic equations. Qualitative models are given to qualitative simulators such as QSIM [Kuipers, 1986], and the simulators produce the qualitative behaviors of the system being modelled.

Traditionally, qualitative models have been constructed by hand. This is workable for simple, well-understood systems. For complex systems, the approach of compositional modelling [Falkenhainer and Forbus, 1990] allows a system model to be built up from predefined components. Although this makes constructing models easier, it still requires the user to understand the system being modelled. Often, however, users want a model precisely because the target system is not well-understood.

This leads to the approach of building a qualitative model automatically, purely from observations of a systems behavior. [Coiera, 1989] presents the beginnings of such a method; given a qualitative description of one or more system behaviors, he derives a qualitative model that reproduces those behaviors. MISQ, an independently developed system presented in [Richards, Kraan, and Kuipers, 1992], uses some of the same techniques as Coiera, but can synthesize qualitative models from qualitative or quantitative behavioral data. MISQ learns maximally constrained models and can handle incomplete behavioral descriptions.

FORTE uses components of MISQ to provide the domain knowledge it needs to work in the domain of qualitative modelling. However, FORTE substantially extends the capabilities of prior versions of MISQ[5], by allowing the invention of new system variables. Results in the domain of qualitative modelling appear in Chapter 6.

[5]We use MISQ as the general name applied to a set of techniques. MISQ has been implemented in a special-purpose system (see [Kraan, Richards, and Kuipers, 1991]) and via FORTE (as described in this thesis and in [Richards, Kraan, and Kuipers, 1992]).

Chapter 3
OBJECTIVE AND FORMAL MODEL

3.1 OBJECTIVE

The objective of this research has been to develop methods for revising first-order theories, and to test these methods by implementing them in the system FORTE. FORTE represents theories internally in first-order Horn clause logic, specifically, in Prolog. FORTE can work with non-logic domains through translator modules that convert between the foreign domain and a first-order logic representation. In addition, FORTE allows domain-specific knowledge to be introduced into the theory revision process. The paragraphs below define our terminology, and provide a more formal statement of this objective.

3.1.1 Theory

A theory is a set of function-free definite program clauses.[6] Forte views theories as pure Prolog programs. In a family domain, for example, a theory would be a set of clauses defining family relationships.

3.1.2 Concept

A concept is a predicate in a theory for which examples appear in the training set. Concepts need not be disjoint.[7] In a family domain, concepts might include father, aunt, and nephew.

[6]A definite program clause is a clause of the form $A \leftarrow B_1, \ldots, B_n$, where $A, B_1, \ldots, B_n$ are atoms [Lloyd, 1987]. Definite program clauses cannot contain negation. Adding negation does not increase the expressive power of logic programs.

[7]Some machine learning systems require concepts to be disjoint, or exclusive. This means that, if we view predicates as relations, the intersection of the relations for different concepts must be empty. For example, a medical system requiring disjoint concepts would be unable to correctly diagnose a patient who had two simultaneous disorders.

3.1.3 Instance

An instance is a ground instantiation of a concept. For example, an instance for the family-relation concept father might be father(frank, susan). A positive instance is one that is true (e.g., Frank is actually Susan's father), and a negative instance is one that is false.

3.1.4 Example

An example is a tuple {P, N, F}, where P is a set of positive instances, N is a set of negative instances, and F is a set of unit clauses (also called facts). The positive instances should be derivable from the theory augmented by the facts as additional axioms; the negative instances should not. In the case of a family domain, the facts would define a particular family, e.g., parent(frank, susan), gender(frank, male), gender(susan, female).

3.1.5 Correctness

Given a set, *P*, of positive instances and a set, *N*, of negative instances, we say theory *t* is *correct* on these instances if

$$\forall p \in P\colon\ t \cup F \vdash p$$
$$\forall n \in N\colon\ t \cup F \nvdash n$$

where *F* is the set of facts associated with *P* and *N*. The set $P \cup N$ is *consistent* if $P \cap N = \varnothing$. A theory can never be correct on an inconsistent set of instances.

3.1.6 Objective

Given an initial theory and a consistent set of instances, produce an "appropriately revised" theory that is correct on the given instances.

3.1.7 Discussion

What is an "appropriately revised" theory? A correct theory can be produced trivially by deleting any existing clauses and asserting new clauses that correspond strictly with individual positive instances, but such a theory is unlikely to be of interest. We say that a theory is appropriately revised if it meets certain heuristic criteria, namely

— A revised theory should be semantically and syntactically similar to the initial theory.

— A theory should be as simple as possible.

— A theory should make meaningful generalizations from the input instances.

Similarity to initial theory. FORTE attempts to keep the revised theory both semantically and syntactically similar to the original theory. The semantic content of a theory is its deductive closure. The syntax of a theory is its structure, i.e., the hierarchy of rules it contains and the antecedents present in each rule. Semantic and syntactic similarity are often competing goals; a small syntactic change may cause a dramatic semantic change. For example, adding a single antecedent to the top-level predicate in a theory can cause the deductive closure of the theory to be empty.

In order to preserve the semantics of a theory, FORTE restricts its revisions to parts of the theory that are known to cause errors. In order to preserve the structure of a theory, many of FORTE's revision operators modify existing rules rather than constructing new ones. In practice, it is almost always easier to modify an existing, partially correct clause than to construct a new one.

Simplicity. In order to keep a theory as simple as possible, FORTE scores revisions both on how accurate they are and on how large they make the theory (the size of a theory is the total number of literals present in the theory). If two possible revisions are equally accurate, FORTE chooses the one that keeps the theory the smallest.

Meaningful generalizations. The bias towards small theories also helps FORTE make meaningful generalizations. Without a bias to keep the theory small, it would be possible to create a correct theory by adding a new clause for each positive instance. A preference for a small theory is also a preference for general rules that cover many positive instances. The quality of generalizations can be empirically measured by testing the revised theory on novel instances taken from the same distribution.

3.2 FORMAL MODEL

This section presents the formal ideas underlying Forte's basic algorithm. The objective of the algorithm is to derive a series of theories that are increasingly accurate on the training set. Forte uses a hill-climbing algorithm that is guided by a global heuristic based on its accuracy on *all* training instances. This is in contrast to the possibility of using a heuristic based on only a *single* instance (e.g., each iteration of back-propagation alters the settings of a neural network based on the errors generated by a single instance).

3.2.1 Definitions

Let T be a space of possible theories, I an instance space (viz., the training set), and $f: I \mapsto \{true, false\}$ a function mapping instances to a boolean value such that positive instances map to *true* and negative instances map to *false*. Let P be a program (viz., FORTE) that revises any theory $t \in T$. The goal of P is to revise t in such a way that, given any $i \in I$, t predicts $f(i)$ correctly.

Program *P* predicts *f*(*i*) by attempting to deduce *i* from *t*, i.e., if *i* is a logical consequence of *t*, *P* predicts *f*(*i*) = *true*, and if *i* is not a logical consequence of *t*, *P* predicts *f*(*i*) = *false*. We say program *P*, and hence theory *t*, is correct on *i* if

$$t \cup F_i \vdash i \text{ iff } f(i) = \mathit{true}$$

and

$$t \cup F_i \nvdash i \text{ iff } f(i) = \mathit{false}$$

which we denote as *correct*(*t*, *i*).

3.2.2 Two models of learning

Let *P*(*t*) represent the set of possible theories *P* can develop by performing one revision on theory *t*. We say *P* learns (i.e., improves the accuracy of *t*) if the following two conditions hold:

$$\forall t \in T, \forall t' \in P(t): \quad \forall i \in I: \; \mathit{correct}(t, i) \rightarrow \mathit{correct}(t', i) \tag{1}$$

$$\begin{aligned} \forall t \in T: \quad & \exists t' \in P(t), \exists i \in I: \; \neg \mathit{correct}(t, i) \wedge \mathit{correct}(t', i) \\ & \vee \forall i \in I: \; \mathit{correct}(t, i) \end{aligned} \tag{2}$$

The first condition guarantees that *P* does not lose knowledge as it executes, i.e., if a theory *t* is correct on some instance *i* then any revision *t'* will also be correct on *i*. The second condition states that it is possible for *P* to execute in a way that leads to an improvement in its accuracy, unless *P* has converged to perfect accuracy. In other words, if a theory *t* is not perfectly accurate then some *t'* will be correct on an instance where *t* was incorrect. While these two conditions form an intuitively pleasing definition, a more general approach can be represented by the single condition

$$\forall t \in T\text{: } \exists t' \in P(t)\text{: } |\{i \in I \ni correct(t, i)\}| < |\{i \in I \ni correct(t', i)\}| \quad (3)$$
$$\vee \forall i \in I\text{: } correct(t, i)$$

This condition says that, with each change of program state, the program is a more accurate predictor of $f(i)$ than it was before (i.e., t' is correct on a larger number of instances than t). This may be less cognitively plausible, since it allows P to lose arbitrary amounts of knowledge, but it imposes fewer restrictions on the way P operates. Forte is based on the approach defined by (3).

3.2.3 Accuracy of Theories

We define accuracy as the number of instances for which a theory predicts the correct value for $f(i)$. We say theory t' is more accurate than theory t iff

$$|\{i \in I \ni correct(t', i)\}| > |\{i \in I \ni correct(t, i)\}|$$

We denote this as

$$t' >_a t \quad (4)$$

Finally, we say a theory is *completely accurate* if

$$\forall i \in I\text{: } correct(t, i)$$

This corresponds to (3) above.

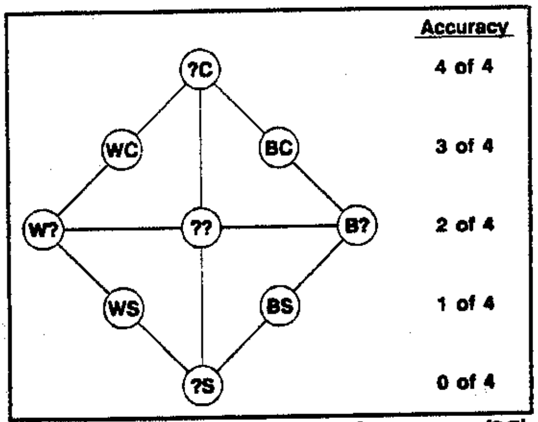

Figure 1. Accuracy graph for the concept '?C', meaning a circle of any color.

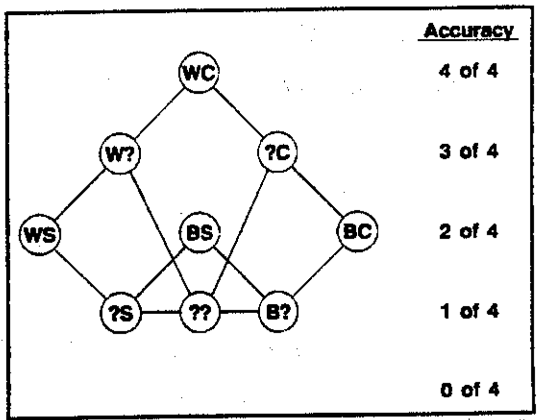

Figure 2. Accuracy graph for the concept WC, meaning a white circle.

Using accuracy as an ordering relation, the theories describable in the theory language form nodes in a directed graph, and the edges represent possible theory revisions. The goal of a learning system is to move upwards in this graph, ideally reaching the top where the system can predict *f*(*i*) with perfect accuracy.

As an example, consider the language consisting of a conjunctive description using the two attributes: {*black*, *white*} and {*square*, *circle*}, and let the instance space contain the four instances: *black-square*, *white-square*, *black-circle*, and *white-circle*. Theories are conjunctive, and there are two revision operators: change a specific value to a variable, and vice versa (hence all edges in this example are bi-directional). Hence, two sample theories in this domain would be

positive ← color(black) ∧ shape(square)
positive ← color(white) ∧ shape(?)

For convenience, we can refer to a theory by using the first letter of each attribute. These two sample theories would be abbreviated BS and W? respectively.

Altogether there are nine possible theories. Figure 1 shows these nine theories arranged as an accuracy graph for the concept *circle*. The theory correctly describing the concept has an accuracy of 4, meaning that it correctly classifies all four instances. The other theories range in accuracy from 0 to 3. The edges represent the possible transitions from one theory to another using the two revision operators. Note that there is a direct upwards path from any theory to the correct theory.

```
Algorithm forte($t$, $t'$)
  let $C$ be the instances on which $t$ is correct
  let $I$ be the instances on which $t$ is incorrect
  generate_revisions($C$, $I$, $t$, $R$)
  choose the best $r \in R$
  if $r(t) >_a t$ then
    forte($r(t)$, $t'$)
  else /* done */
    $t' = t$
  end if
end forte

Algorithm generate_revisions($C$, $I$, $t$, $R$)
  for each $op \in OP$
  create a revision $r_{op} \in R$
end generate_revisions
```

Figure 3. Top-level revision algorithm.

Figure 2 shows an accuracy graph for the concept *white-circle*. This graph contains a local maximum. If the initial theory is *black-square*, all possible revisions decrease the accuracy of the theory. If a learning system is to avoid being trapped by such maxima it must be able to either escape from them or avoid them in the first place (e.g., Version Space in [Mitchell, 1982] avoids local maxima by maintaining multiple concepts).

Although any hill-climbing algorithm can be trapped by local maxima, FORTE reduces its vulnerability by defining a wide variety of revision operators. Adding operators adds edges to the theory graph, and thereby eliminates many local maxima.

3.2.4 The FORTE Algorithm

The basic FORTE algorithm is a straightforward application of (3). To make a transition from t to $t' \in P(t)$ that satisfies (3), FORTE looks for revisions it can apply to t to create theory t' such that

$t' >_a t$

If there are several such revisions, it chooses the best one (see below). This process iterates until there is no revision that improves the accuracy of the theory. At this point, either t' is completely accurate or Forte is trapped in a local maximum.

This algorithm is shown in Figure 3. We can examine this algorithm in two parts: the hill-climbing that selects a revision to implement, and the search strategy that composes revisions and leads to a final revised theory.

Hill-climbing. The algorithm *generate_revisions* represents the fact that FORTE has a library of several revision operators. When FORTE generates revisions, every operator is asked to generate one or more possible revisions to the theory. All of these revisions are collected, and the best (see below) is selected for implementation. In essence, the operators are competing to produce the best revision.

Revisions are evaluated on two criteria: accuracy and simplicity. The revision that produces the most accurate theory is always preferred. If there is more than one revision that produces the same accuracy increase, the one leading to the simplest theory (see Section 3.1.7) is selected.

Search Strategy. There are a number of ways to search the space of possible theories. FORTE uses the most efficient one possible: depth-first search without backtracking. This strategy will only produce accurate theories if local maxima do not pose a significant problem. Since we have a known goal-state (100% accuracy) backtracking would allow Forte to always escape local maxima, although at a high cost in efficiency. In practice, the variety and sophistication of revision operators that FORTE uses serve to eliminate most local maxima in the theory space, thus allowing us to dispense with backtracking (see Chapter 6 for empirical verification of this point).

Learning a theory that is accurate on the training instances is only useful if the theory generalizes well to unseen instances. When learning in a finite hypothesis space[8], PAC learning theory[9] [Haussler, 1988] guarantees that learning an accurate theory on a training set does lead to learning on unseen instances. The hypothesis spaces of many first-order domains are finite. In a domain where the hypothesis space infinite, the user may be able to establish some upper bound on the size of a correct theory. This upper bound serves to make the hypothesis space finite, since the theory language contains a finite number of symbols. Given a finite hypothesis space of size H, the number of instances needed to meet PAC requirements is $O(\log |H|)$. For languages whose expressiveness is limited, the limit may be much lower.

[8]The *hypothesis space* is the set of distinct concepts expressible in the theory language, ignoring semantically equivalent theories.

[9]PAC stands for *probably approximately correct*. PAC learning theory analyzes the probability that a learned concept approximates the target concept to a specified degree of accuracy.

Chapter 4
FORTE OVERVIEW

This chapter presents an overview of FORTE. The first section looks at FORTE's interface to the outside world, including the training examples, the initial theory, and any available domain knowledge. The second section examines the theory revision process itself—how FORTE specializes, generalizes, and compacts clauses in a theory. However, detailed algorithms are left to Chapter 5.

4.1 EXTERNAL INTERFACES

Figure 4 shows FORTE's interface to the outside world. FORTE itself is represented by the center box. The remaining boxes represent auxiliary modules that vary from domain to domain. They provide for user convenience when working with domains whose native representation differs from that used by FORTE, they can help limit the search space FORTE must explore, and in some domains, they can substantially enhance FORTE's ability to develop good revisions. All of the auxiliary modules are optional and can be omitted. In addition to the auxiliary modules shown, the user may specify an explicit language bias.

4.1.1 Revision verifier

The revision verifier provides a way to introduce domain-dependent knowledge into the revision process. At various stages of the revision process, FORTE calls the revision verifier to ensure that it is creating a potentially useful revision. If the user has no knowledge of special heuristics or requirements in the domain, the revision verifier may be omitted.

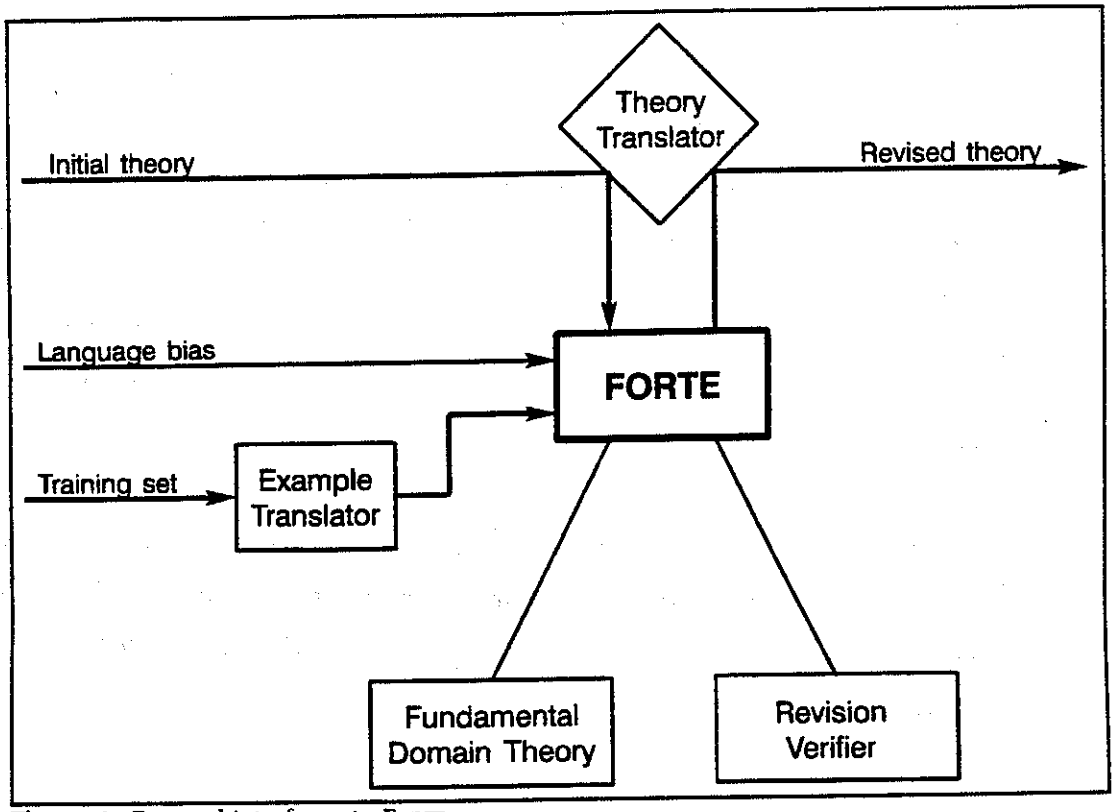

Figure 4. External interfaces to FORTE.

The revision verifier allows the user to insert domain-specific consistency checks three places in the revision process. First, the revision verifier can reject data elements constructed by FORTE during the revision process. This is often useful in domains where atoms contain implicit function symbols (e.g., when working with lists), as FORTE cannot itself tell when a constructed value is invalid. For example, if FORTE is revising a predicate to work with lists, the training set may only contain lists of length four and less. If FORTE adds a literal that would create lists of length eight (e.g., append), the revision verifier can reject such lists as invalid, and FORTE will abort the revision. The revision would probably be rejected anyway, since it would not correspond to any instances in the training set, but doing so early can save substantial amounts of work.

Second, the revision verifier may be passed groups of antecedents intended to appear together in a clause. If it detects any inconsistency in this set, it may reject it. An example of this is the dimensional analysis performed by the revision verifier for qualitative modelling. Suppose FORTE plans to add the contraints derivative(X, Y) and add(X, Y, Z) to the same clause. The derivative constraint requires X and Y to have different dimensions, while the add constraint requires that their dimensions be the same. Consequently, the revision verifier rejects this combination, and FORTE rejects the corresponding revision.

Finally, the revision verifier is given the chance to review complete revisions before FORTE evaluates them for accuracy and simplicity. If the revision verifier detects an inconsistency in a revision, it can reject it and FORTE will delete it from the set of revisions being considered for implementation.

4.1.2 Fundamental domain theory

The fundamental domain theory serves two purposes. First, it provides a place for theory predicates which are known to be correct, and which FORTE should not revise. For instance, when working with lists, the user might wish to provide standard definitions for member/2[10] and append/3. Since FORTE does not revise predicates in the fundamental domain theory, these predicates may be written using all of the features of Prolog (e.g., functions, built-in predicates, and cut). Second, the fundamental domain theory can provide complex definitions for the fundamental relations used to define a

[10]When discussing first-order theories, we use Prolog notation for predicates. Thus, member/2 refers to a predicate named "member" that takes two arguments, e.g., member(X, Y). Also in keeping with Prolog notation, ":-" is read as an implication arrow pointing to the left, and variable names are capitalized.

domain. Normally, the facts in an example are taken as extensional definitions[11] of the corresponding relations appearing in the theory. However, the user may wish to provide more sophisticated definitions. For example, when working in a family domain, one of the fundamental relations which may be used to define a family is married/2. The fundamental domain theory might provide the following definition for this relation:

```
married(X, Y) :- example(married(X, Y)), !.
married(X, Y) :- example(married(Y, X)).
```

where example(married(X, Y)) is a reference to a fact defined in an example. This reflects the fact that marriage is a commutative relation.

4.1.3 Theory translator

The theory translator is an optional module used to translate between the native representation of a theory and the representation required by FORTE. The most common use of the theory translator is to eliminate function symbols from the theory. Theories in FORTE are represented as function-free definite clauses, and function symbols which appear in instances must be explicitly interpreted by predicates in the theory.

As an example, when working with lists, it is more convenient for the user to write and examine theories containing function symbols, i.e.,

```
append([A|B], C, [A|D]) :- append(B, C, D).
```

[11] An extensional definition is one that explicitly provides every true instantiation. For example, in a family domain, parent(X, Y) is assumed to be true for exactly those pairs listed as facts in the examples (e.g., parent(frank, susan), parent(alice, fred), and so forth).

The theory translator for list domains converts the functional list construction into a predicate construction using components/3. Thus, FORTE would see the equivalent clause

```
append(X, C, Y) :-
        components(X, A, B), components(Y, A, D),
        append(B, C, D).
```

After FORTE has revised the theory, the theory translator is called to convert the result back into functional format.

4.1.4 Example translator

FORTE requires examples to be provided as complex Prolog terms. As with theories, the FORTE representation may not be convenient in all domains. The example translator can be used to translate between a native domain representation and that required by FORTE.

4.1.5 Language bias

Another way that domain knowledge can be provided to FORTE is through an explicit language bias. The language bias contains a number of elements. First, it tells FORTE what antecedents are allowed in a theory. For example, antecedents may include calls to theory predicates, calls to fundamental relations used to define the domain, or calls to built-in equality predicates. This allows the user to restrict the amount of search done by restricting the theory space. If in doubt, the user simply allows all possible antecedents.

Second, the user may select the options conjunctive and most-specific. A conjunctive theory is one where there is only a single clause for each predicate. A most-specific theory is a conjunctive theory where each rule is made a specific as possible—this allows FORTE to learn useful theories in the absence of any negative instances. These options provide a very strong theory bias, and should only be used if the theory is known to be conjunctive.

Third, the user may allow or disallow recursive predicates in the theory. If the user is not certain that the desired theory is non-recursive, the user should allow recursion.

Fourth, the language bias allows the user to set the maximum proof-depth allowed when attempting to prove instances using the theory. FORTE uses this depth limit to prevent looping in the prover, as well as to limit other search processes in the revision operators. Generally speaking, the user can set a large limit and be unconcerned with its effect. However, setting relatively tight limits when working with recursive theories (which are prone to looping) can make the system substantially more efficient.

Finally, the language bias includes one explicit tuning parameter. The other elements of the language bias serve to restrict the theory space that FORTE must search, but the user can choose "least restrictive" settings. This last parameter, relation-tuning, has no "least restrictive" setting. Instead, it provides three distinctly different biases to the way FORTE develops revisions: highly-relational, relational, and non-relational. Details on the effects of these choices are given in Chapter 4, but a summary is presented here.

Different biases are suited to different domains. Domains where relations are characterized by relational paths (see *relational pathfinding* in Chapter 4), should use a setting of "highly-relational." This is appropriate, for example, for family relationships. Selecting any other setting will result in less accurate revised theories. If a domain is essentially propositional, then a setting of "non-relational" should be used. The intermediate setting, relational, is the default, and should be used for all other types of domains.

4.2 REVISING THEORIES

The top-level algorithm for FORTE presented in the last chapter is a hill-climbing algorithm. Each iteration FORTE generates a set of possible revisions, the best of which is chosen and implemented. This process repeats until no further revisions are possible. This section provides an overview of the methods FORTE uses to generate revisions; specific algorithmic details are presented in the next chapter.

FORTE performs three types of revision on theories: specialization, generalization, and compaction. Specialization and generalization are performed in hopes of improving the theory's accuracy. Revisions generated are ranked by how much they improve theory accuracy, and secondarily by their simplicity. Compaction revisions are only generated when no specialization or generalization revision improves the theory's accuracy.

The examples in this section are cast in two domains. The first of these is the domain of family relationships in the family shown in Figure 5. This is one of the families in Hinton's family data [Hinton, 1986]. The second domain is a blocks-world consisting of tables (of various colors) and blocks (of various shapes and colors). In this domain, we seek to describe a "likeable" configuration of tables and blocks, based on color, shape, and stacking (i.e., which blocks are on which table).

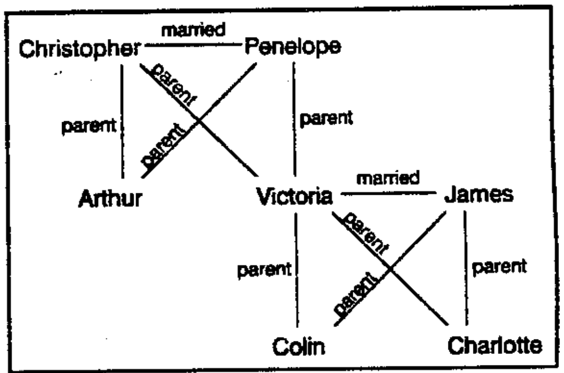

Figure 5. A family from Hinton's family data.

4.2.1 Specializing predicates

FORTE specializes predicates when negative instances are provable by the current theory. A target clause may be specialized by being deleted, or by having antecedents added to it. Deletion is a simple operation; the clause is removed from the theory, and all positive and negative instances whose proofs made use of the clause are reproven. If the result is a more accurate theory, then deletion is a possible revision.

On the other hand, deletion is fairly drastic, and likely to make positive as well as negative instances unprovable. It is much more likely that a clause can be appropriately specialized by adding new antecedents to it. We may actually need to produce several specialized clauses in order to cover all of the positive examples. For example, in the blocks-world domain, suppose we like round blocks that are either red or green. If we begin with the rule

```
likeable(X) :- block(X), shape(X, round).
```

then we must specialize this rule in two separate ways to obtain the correct theory

```
likeable(X) :- block(X), shape(X, round), color(X, red).
likeable(X) :- block(X), shape(X, round), color(X, green).
```

In order to specialize a clause, FORTE adds antecedents to it to make all negative instances unprovable. The goal is to keep as many positives provable as possible, but it may not be possible to retain proofs for all of them. In this case, FORTE adds the new specialization to the theory and begin again with the original clause, looking for alternate specializations that will retain the proofs of the other positive instances. This process repeats until we have a set of clauses that retains the provability of all of the originally provable positive instances.

FORTE provides two separate algorithms for producing a specialized clause, called hill-climbing antecedent addition and relational pathfinding. The interaction of these two algorithms is determined by the relation-tuning parameter in the language bias. If the relation-tuning is set to "highly-relational" then clauses developed by relational pathfinding are always preferred. If relation-tuning is set to "non-relational" then relational pathfinding is not used.

When relation-tuning is set to "relational" (or omitted, since this is the default), both hill-climbing antecedent addition and relational pathfinding develop clauses, and the clause with the best performance (accuracy and simplicity) is selected. In practice, these two methods of specializing clauses are complementary; certain types of revisions are performed well by one but not the other.

The hill-climbing algorithm is reminiscent of FOIL. FORTE considers all antecedents that could be added to the current clause, scoring each on its ability to discriminate positive and negative instances. It selects the best antecedent and adds it to the clause. This process continues until all negatives have been eliminated or until no antecedent provides any gain. As an example, suppose we wish to define the grandfather relation, beginning with the overly-general rule

```
grandfather(X,Y) :- gender(X, male).
```

and using one positive and two negative instances:

+ grandfather(christopher, charlotte)
- grandfather(james, charlotte)
- grandfather(colin, charlotte)

One of the antecedents considered is parent(X,Z), which states that X has a child. This antecedent provides a gain since adding it eliminates the instance grandfather(colin, charlotte). However, it does not eliminate the other negative instance, so we continue specializing. The best antecedent to add next is parent(Z,Y), since it eliminates the remaining negative instance while still allowing the positive instance to be proven.

The discrimination measure used to score antecedents is the information theoretic measure used by FOIL. However, FORTE considers only the number of provable positive and negative instances, whereas FOIL's tuple-based approach counts the number of *proofs* of positive and negative instances. FORTE's approach is more efficient since it does not compute all proofs, but results in a slightly different bias.

The hill-climbing approach is quite effective in many cases, particularly for developing recursive base-cases and for adding non-relational antecedents to a rule. However, as with any hill-climbing method, it can be caught by local maxima. There is also another type of locality problem, called a *local plateau*, where there are many antecedents that do not decrease accuracy, but in order to actually increase accuracy one must add several antecedents at once.

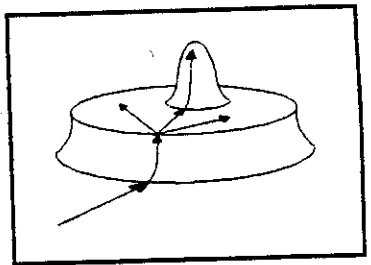
Figure 6. A local plateau.

We can see the local plateau problem by trying to define the grandparent relation using only the instances

+ grandparent(Christopher, Colin)
- grandparent(Christopher, Arthur)

There is no single antecedent that we can add which will allow the positive instance to be proven while making the negative instance unprovable. Both Colin and Arthur have parents, neither has children, and neither is married. Even determinate literals do not help in this example, since all parents have two children and all children have two parents. In order to create a correct theory, we must simultaneously add both of the required parent relationships, i.e.,

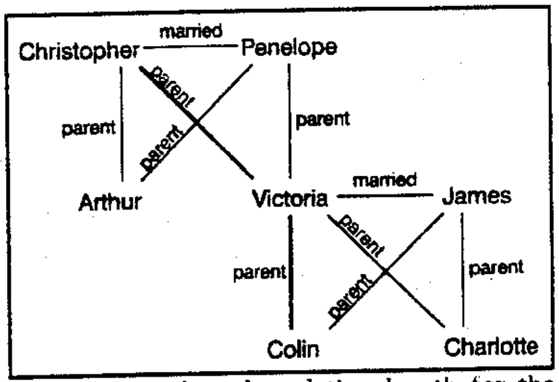

Figure 7. Finding the relational path for the grandfather relation.

$$grandparent(x, y) \leftarrow parent(x, z) \wedge parent(z, y).$$

Relational pathfinding is able to do this since it works on the basis of graph-search, seeking the shortest path of relations joining the constants christopher and colin in the positive example.

Relational pathfinding [Richards and Mooney, 1992] is based on the assumption that, in relational domains, the ground atoms defining a positive instances are usually linked by a short fixed path of relations. In the example above, the grandfather relation is characterized by a fixed path containing two parent relations. To find such a path, we view the domain as a (possibly infinite) graph of constants linked by the relations that hold between them.[12] We locate the nodes corresponding to the constants of a positive instance, and explore paths from all nodes simultaneously until we find an intersection. If the positive instance chosen for relational pathfinding was representative of many positive instances, the resulting specialization will discriminate well between the positive and negative instances.

4.2.2 Generalizing predicates

FORTE generalizes a predicate when a positive instance is unprovable. It uses four operators to perform generalization. Two methods are similar to methods used in propositional theory revision: adding new rules and deleting antecedents from existing rules. The second two are variants of the inverse-resolution techniques absorption and identification.

In order to add a new rule, FORTE first copies the existing (overly-specialized) rule and deliberately overgeneralizes it. It does so by deleting all antecedents whose deletion either does not allow any negatives to become provable or whose deletion allows one or more positives to become provable. This reduces the clause to a core of essential antecedents. FORTE passes this overly-general rule to the antecedent addition algorithm described in Section 4.2.1, which specializes the rule to eliminate provable negatives. This process may create several rules, all of which will be added to the theory.

However, FORTE may be able to create a good revision simply by deleting antecedents from an existing clause. It includes two different methods to do this, just as there were two methods to add antecedents to a clause; one is a simple hill-climbing method, while the other is well-suited to

[12]For many domains, like that of family relationships, this is a natural representation. However, any domain can be represented in this way. If we have relations with arity greater than two, they are simply edges with more than two ends.

escaping local maxima and local plateaus. Both methods may create several different generalizations of the original clause, in order to make as many positives provable as possible while still excluding the negatives.

The hill-climbing method of antecedent deletion evaluates all antecedents on the basis of how many positives would be provable if the antecedent were deleted. It does not consider any antecedent whose deletion would allow proof of a negative instance. It deletes the best antecedent and recurses. This process terminates either when all positives are provable, or when no further antecedents can be deleted without allowing proof of a negative.

This simple approach to deleting antecedents may be unable to create any useful generalizations of the original clause. For example, it may be necessary to delete at least two antecedents before any positive will become provable; this is another version of the local plateau problem. If this occurs, FORTE uses a different method, which deletes multiple antecedents.

The method of deleting multiple antecedents is a general one, which can solve arbitrary "m of n" problems.[13] The algorithm uses a guided, exhaustive depth-first search process, deleting any antecedents it can without allowing proof of any negatives. Once it has deleted all antecedents possible, it checks to see if one or more positive instances have become provable. If so, the newly generalized clause is returned. If not, the algorithm backtracks,

[13]An *m of n* problem is one where any subset of the given antecedents is sufficient. For example, suppose the initial rule is: `concept :- a, b, c, d, e.` If positives instances may satisfy any three of the given antecedents, and no negative will satisfy more than two, then a correct theory consists of ten rules of the form:

```
concept :- a, b, c.  concept :- a, b, d.  concept :- a, b, e.
concept :- b, c, d.  concept :- b, c, e.  concept :- b, d, e.
```

undoing a previous deletion and searching for another combination of antecedent to delete. This algorithm is described in detail in Chapter 5.

FORTE's third method of generalization is the inverse resolution operator identification. Identification seeks to construct an alternate definition for an antecedent that is failing in attempted proofs of positive instances. It develops an alternate definition by performing an inverse resolution step using two existing rules in the theory. For example, suppose a call to predicate x is the failing antecedent. We thus would like to generalize our definition of x, and we have the following two rules in the domain theory:

$a \leftarrow b, x$
$a \leftarrow b, c, d$

Identification will replace these two rules with the equivalent pair:

$a \leftarrow b, x$
$x \leftarrow c, d$

While this has no effect on the deductive closure of these rules alone. we have now introduced an additional clause for x, and have thus generalized its definition.

The fourth method of generalization is the inverse resolution operator absorption. Absorption is the complement of identification. Rather than constructing new definitions for intermediate predicates, absorption seeks to allow existing definitions to come into play. Suppose predicate c in the rule below is a failure point:

$a \leftarrow b, c, d$ (1)

Now suppose the theory contains the following rule, as well as other rules with consequent x:

x ← c, d

In this case, absorption would replace rule (1) with the new rule

a ← b, x

thereby possibly allowing alternate definitions of x to be used when proving predicate a.

4.2.3 Compacting Predicates

When FORTE is unable to create any useful specializations or generalizations of the theory, it attempts to compact it. The compaction operators are designed to change the deductive closure of the theory as little as possible. However, if FORTE is caught in a local maximum, compaction may alter the theory in a way that allows it to escape. Otherwise, the compaction factors out some common antecedents, thereby making the theory more readable.

Revisions proposed by compaction operators are scored in the same way as revisions proposed by specialization and generalization operators: on accuracy and simplicity. Compactions are not expected to increase the accuracy of the theory, but this scoring prevents them from decreasing accuracy, and ensures that the most beneficial compactions are implemented first.

Compaction uses two operators: identification and absorption. These are similar to the generalization operators of the same names, but are driven by a requirement to reduce the size of the theory, rather than to increase its accuracy. For example, if we have the theory

```
a :- b, c, d
a :- b, e, f
x :- e, f
```

we can reduce its size using absorption

```
a :- b, c, d
a :- b, x
x :- e, f
```

and again by using identification, thus yielding the compacted theory

```
a :- b, x
x :- c, d
x :- e, f
```

Of course, this may generalize the theory, so we only perform these compactions if doing so does not reduce accuracy on the training set.

Chapter 5
THEORY REVISION ALGORITHMS

This chapter presents the details of FORTE's theory revision algorithms. The first section discusses the algorithms that drive the overall revision process, using the revision operators to develop revisions. The second section explains special provisions that FORTE makes for recursive and most-specific theories. The third section gives the algorithms for the revision operators themselves. Finally, the last section briefly discusses the computational complexity of FORTE.

5.1 TOP-LEVEL ALGORITHM

FORTE revises theories iteratively, using a hill-climbing approach. Each iteration identifies points in the theory, called *revision points*, where a revision has the potential of improving the theory's accuracy. It then generates a set of revisions, selects the best one, and implements it. This process continues until no revision improves the theory. This top-level algorithm is shown in Figure 8.

In order to generate revision points, the current theory is tested on the training set. FORTE annotates failed proofs of positive instances and successful proofs of negatives. From these annotations it identifies points in the theory for possible revision (this is discussed in more detail in Section 5.1.1). The revision points are sorted according to their potential—the maximum increase in theory accuracy that could result from a revision of that point. For example, if a particular clause was used in successful proofs of five negative instances, then specialization of that clause has a potential of five.

FORTE then generates a set of proposed revisions from the revision points, beginning with the revision point that has the highest potential and working down the list (see Section 5.1.2). We stop generating revisions when the potential of the best remaining revision point is less than the accuracy increase gained by the best revision generated to date. FORTE scores revisions on the change they make to the theory's accuracy on the training set and on the effect they have on the size of the theory. A revision is said to improve the theory if it either increases accuracy or has no effect on accuracy but reduces the theory's size. The revision that best improves theory accuracy is selected; in case of a tie, the revision that makes the theory the smallest is preferred. For example, if we have the revision scores

```
Repeat
  Generate revision points
  Sort revision points by potential
  For each revision point, best to worst
    Generate revisions
    Update best revision found
  Until potential of next revision point
      < benefit of best revision to date
  Implement the best revision
until no revision improves the theory
```

Figure 8. The top-level FORTE algorithm.

score(3, -4)	improves accuracy by three instances increases size of theory by four literals
score(2, 1)	improves accuracy by two instances decreases size of theory by 1 literal
score(2, -6)	improves accuracy by two instances increases size of theory by six literals

we would order them as shown, and select the top-most revision. The best revision is implemented, and the cycle begins again.

This process continues until a cycle produces no revisions that improve the theory. At this point, we hope to have developed a theory that is correct on the training set. However, since this is a hill-climbing process, FORTE can be caught in local maxima. We minimize this danger in two ways. First, revisions are developed and scored using the entire training set, rather than just a single instance; this global vision gives us better direction than we would have if revisions were developed from single instances. Second, FORTE uses a variety of different operators to generate possible revisions. Since the operators have different strengths and weaknesses, they are able to escape different types of locality problems.

5.1.1 Generating revision points

Revision points are places in a theory where we suspect that errors may lie. They are of two types: specialization points and generalization points. We identify revision points by annotating proofs or attempted proofs of misclassified instances.[14] Points in the theory where proofs of positive instances fail are places where we may want to generalize the theory. Rules used in successful proofs of negative instances are points where we may want to specialize the theory. The same revision point may be flagged by several different instances, and the number of such instances represents its potential—the maximum increase in theory accuracy that we could possibly gain by revising the theory at that point.

Generating specialization revision points is simple. We simply note which clauses participate in proofs of negative instances, and these clauses become revision points. The algorithm is given in Figure 9.

```
For each provable negative
  note all clauses in the successful proof
end for
For each clause in the theory
  potential =
    number of incorrect proofs using it
  if potential > 0
    record a specialization point
  end if
end for
```

Figure 9. Generating revision points for specialization.

Generating revision points for generalization is more complex because we have three kinds of generalization operators. Some generalization operators are antecedent-based, meaning that

[14]All proofs are carried out using depth-first SLD resolution with backtracking. This is the standard Prolog proof technique. A depth bound is used to prevent looping.

their revisions target a particular antecedent in a particular clause, some are clause-based, and some are predicate-based. We need a revision point for each of these three operator types. However, all of these revision points are generated from annotations made during attempted proofs of failed positive instances. The algorithm appears in Figure 10.

```
For each unprovable positive
  mark all failing antecedents in the proof-tree
  mark all antecedents that may contribute to the failure
end for
For each marked point in the theory
  potential = number of instances marking it
  if potential > 0
    record antecedent-based generalization point
  end if
end for
For each clause with a marked point
  potential = number of instances marking a point in the clause
end for
For each predicate in the theory referenced by a marked point
  potential = number of instances marking a call to the predicate
end for
```

Figure 10. Creating revision points for generalization.

The annotation process itself is fairly complex. Each time we are forced to backtrack, we note which antecedent in which clause failed and caused us to backtrack; the failing antecedent is a "failure point". In addition, we must consider which other antecedents may have contributed to this failure, perhaps by binding variables to incorrect values. These antecedents are called "contributing points". As an example, consider the following program, when called with the goal a(1):

```
Program                               Facts
a(Y) :- b(X), cc(Y), d(X,Y).          bb(1).
b(X) :- bb(X).                        cc(1).
d(X,Y) :- dd(X,Y).                    dd(2,2).
```

In this program we have failure points d(X,Y) and dd(X,Y), since these two antecedents fail (i.e., are called but are never successfully proven). Other antecedents, such as cc(Y) are not marked as failure points even though they do fail on backtracking, since they were successfully proven during the original traversal of the proof branch in which they appear.

In addition, we note that X is bound by b(X) and ultimately by bb(X), so we mark these two antecedents as contributing points. We do not mark cc(Y) since it did not instantiate Y. Our final annotations are:

```
a(Y) :- b(X), c(Y), d(X,Y).
b(X) :- bb(X).
d(X,Y) :- dd(X,Y).
```

No distinction is made between failure points and contributing points. These marked antecedents become our antecedent-based revision points. They represent all points in the theory where generalization may improve theory accuracy. We derive the other types of generalization revision points from the antecedent-based revision points.

We create clause-based revision points for all clauses in which we made an annotation. The potential of a clause-based revision point is the number of distinct instances that marked any point within it. These revision points are used by clause-based operators like add-rule, which generalize a clause without regard for any particular antecedent. In the above example we would have three clause-based revision points since all three clauses contain marked points.

Predicate-based revision points are the next step beyond clause-based revision points. A predicate-based revision point is created for each theory predicate that appears as a marked antecedent in the annotated theory. In other words, since we marked b(X) in the theory, we create a predicate-based revision point for b/1. However, we do not create a predicate-based revision point for a/1, since no antecedents referencing a/1 were marked. Predicate-based revision points have a potential equal to the number of distinct instances that annotated a call to the predicate anywhere in the theory. These revision points are used by the operator identification, which seeks to generalize the definition of the predicate, without reference to any particular clause. In the above example, the marked points reference two predicates defined in the theory, b/1 and d/2, so we would have two predicate-based revision points.

5.1.2 Developing revisions

Once revision points have been generated, they are sorted by decreasing potential. FORTE then develops revisions for each revision point in turn, beginning with the revision point that has the highest potential. This process is outlined in Figure 11. For a given revision point, FORTE calls all operators that use that type of revision point to produce proposed revisions to the theory. It continues developing revisions until one of two things happens.

If we reach a point where the best revision generated to date increases theory accuracy more than the potential increase in accuracy of any remaining revision point, FORTE stops and returns the best revision proposed for incorporation into the theory. If, on the other hand, we run out of revision points and still have no revision that improves the theory, FORTE develops additional revisions using the compaction operators. If the theory is completely accurate on the training set, this is the final compaction stage before returning the revised theory. If, on the other hand, the theory is not

```
For each revision point, from best to worst
  Generate revisions
  Update best revision found
Until potential of next revision point
        < accuracy increase of best revision
   or no more revision points
if best revision does not increase accuracy of theory
Generate compaction revisions
  Update best revision found
end if
return best revision
```

Figure 11. Generating revisions.

completely accurate, FORTE is trapped in a local maximum, and the compaction operators may help it to escape.

5.2 SPECIAL PROVISIONS

There are two types of theories, as specified by the language bias, for which FORTE makes special provisions: recursive theories and most-specific theories. These provisions are discussed below.

5.2.1 Recursive theories

Revising a recursive theory is substantially more difficult than revising a nonrecursive one. With nonrecursive theories, we can treat the predicate under revision in isolation from the rest of the theory. If the predicates appearing as antecedents contain slight errors, we will still be able to develop a revision for the chosen predicate. If the antecedents contain gross errors, the proposed revision is likely to simply eliminate them as antecedents. When revising a recursive theory, we inevitably need to evaluate a recursive call to the very predicate we are revising. And, since we are revising it, we can be almost certain that the results of evaluating the recursive call will be incorrect. Furthermore, when a recursive definition is wrong, it is usually catastrophically wrong—misclassifying large numbers of instances due to looping or other problems. And, unlike the nonrecursive case, we do not have the option of simply eliminating the antecedent.

In order to solve these problems, we must decouple our evaluation of a recursive call from the definition of the predicate that we are modifying. The training set provides us with a way to do this; we can use the positive instances in the training set as an extensional definition of the predicate.[15]

[15]FOIL also uses the training set to provide extensional definitions. When FOIL adds an antecedent to a clause, the variables in the antecedent receive values from the defined tuples. The validity of these bindings are checked against the training set. If the antecedent includes

By using this extensional definition to correctly evaluate recursive calls, we allow the revision process to work unhindered by the complications of recursion. And, after the revision has been developed, we can test its actual effectiveness using normal resolution.

Unfortunately, using the training set as an extensional definition works only if the training set is in some sense complete. For example, if we are learning a definition of list reversal, and we wish to prove the example reverse([a,b,c],[c,b,a]), then the training set must contain the examples reverse([b,c], [c,b]) and reverse([c], [c]). If either of these instances is missing, our proofs will fail and we will not be able to develop a correct revision. Hence, when revising recursive predicates, the training set must contain all examples that will be generated during well-founded recursion from other examples present. Since the user is not expected to know what recursion scheme is appropriate for the theory, this means that the training set should contain a complete set of examples below a certain size.

If the recursive predicate we wish to revise is not a top-level predicate for which we have training instances, FORTE derives a temporary training set for the predicate from the top-level predicates. This process works well if the higher-level predicates are correctly defined, but may develop different predicates than expected if the higher-level predicates themselves contain errors.

To see how we derive a training set, suppose we have the (correct) predicate for a subset predicate, i.e.,

new variables, the tuples are extended using all possible bindings found in the training set.

```
subset([], A).
subset([Elt|Elts], Set) :- member(Elt, Set), subset(Elts, Set).
```

and we wish to derive a training set for member. Our training set contains all positive instances for subsets of a set contain three or fewer elements:

```
subset([], []).                    subset([], [a]).
subset([], [b]).                   subset([], [c]).
subset([], [b,c]).                 subset([], [a,c]).
subset([], [a,b]).                 subset([], [a,b,c]).
subset([a], [a]).                  subset([b], [b]).
subset([c], [c]).                  subset([a], [a,b]).
subset([a], [a,c]).                subset([a], [a,b,c]).
subset([b], [a,b]).                subset([b], [b,c]).
subset([b], [a,b,c]).              subset([c], [a,c]).
subset([c], [b,c]).                subset([c], [a,b,c]).
```

and so forth. To derive a training set for member, we collect all of the calls to member made at the top-most level by subset (i.e., we do not descend into the recursive levels, since the results of doing so depend on the correct functioning of member—the predicate we are seeking to revise). We thus have the following correspondence between subset instances and derived member instances

```
     subset([a], [a])  —  member(a, [a])
   subset([a], [a,b])  —  member(a, [a,b])
 subset([b,c], [b,c])  —  member(b, [b,c])
subset([a,b], [a,b,c]) —  member(a, [a,b,c])
```

By collecting all of these instances, we develop a training set for member. The effectiveness of this technique depends on two things. First, if the higher level predicate is not correct, the process may either fail or else define an unexpected relation, which happens to be called member. In the latter case, we will still develop a revision leading to a correct theory; it just may not be the expected theory. Second, the type of calls made to the lower level predicate are important. In the subset example, we generate a complete training set for member. If, however, we used a definition of the reverse predicate

```
reverse([], []).
reverse([A|B], C) :- reverse(B, D), append(D, [A], C).
```

to generate a training set for append, the derived training set would never have more than a single element in the second argument. It would thus define a predicate that adds one element to the end of a list. This predicate is correct in the context of reverse/2, but should not be called "append."

5.2.2 Most-specific Theories

In some domains we wish to make the deductive closure of a theory fit the known positive instances as tightly as possible. These domains are usually ones in which we are learning from positive instances only. In order to prevent simple memorization, where we learn one rule for each positive instance, FORTE insists that most-specific theories also be conjunctive.

An example of a domain requiring a most-specific theory is qualitative modelling. Given a set of input behaviors, we wish to develop a model that reproduces those behaviors. Negative behaviors are not readily available. We could generate an infinite number of them, but which ones we should give to a learning system is not immediately clear. Hence, we ask FORTE to develop the most constrained model that will reproduce all of the input behaviors. Since this is a single model, it will be represented by a single clause (and, hence, it is conjunctive).

In order to develop a most-specific theory, FORTE follows the normal revision process to generalize the input theory as necessary to allow all positives to be provable. It then makes the theory as specific as possible by adding all possible antecedents (which do not eliminate any positive instances) to each clause in the theory. In order to ensure that we add a finite number of antecedents, we do not allow the hill-climbing version of antecedent addition to introduce new variables during this process.

5.3 REVISION OPERATORS

Theory revision operators must be able to transform any theory expressible in the theory language into any other theory in the language. We can do this with the four basic revision operations:

-- adding a rule
-- deleting a rule
-- adding an antecedent to a rule
-- deleting an antecedent from a rule

A simple implementation of four operators to perform these basic operations would produce a workable theory revision system. However, such a system would often find itself trapped in local maxima or lost on local plateaus. Also, the system would be inefficient, since a semantically simple revision may require many applications of the operators. FORTE's operators are designed to avoid most local maxima and local plateaus, as well as to improve efficiency. However, they can often best be understood by remembering that they are ultimately composed of the four basic revision operations.

This section presents the algorithms for FORTE's revision operators. The algorithms are stated in terms of the changes they make to the theory. Recall, however, that each operator is developing a *proposed* revision, and that the revision will be implemented only if it is the best revision developed by any operator for any revision point. The examples in this section are drawn from two domains. The first is a simple blocks-world consisting of tables described by color {red, green, blue} and blocks described by shape {round, square, triangular}. In this domain we seek to describe a "likeable" combination of blocks and tables. The second example domain is Hinton's family data.

Conceptually, each operator develops its revision using the entire training set. However, in practice, this is unnecessary. For example, when specializing a clause using add-antecedent, we will not change the provability of any unprovable instance, or of any provable instance whose proof does not rely on the clause being specialized. Hence, add-antecedent develops its revision using only provable instances whose proofs rely on the target clause.

5.3.1 Delete rule (specialization)

When given a revision point that identifies a clause used in a proof of one or more negative instances, this operator deletes the clause. There are two restrictions. First, if the clause is the only base case of a recursive predicate (i.e., a predicate that currently has one or more recursive clauses), we are not allowed to delete it. Second, if this is the only clause for a concept, we delete the clause, but replace it with the rule

```
if theory is recursive
  if clause is last base case
    and recursive cases exist
    no revision possible
  end if
end if
delete clause from predicate
if predicate is empty
  add the failure rule
end if
```

Figure 12. The delete-rule operator.

```
concept :- fail.
```

This has the same effect on the theory as deleting the rule, but we retain a clause to provide a starting point for later revisions to this predicate.

5.3.2 Delete antecedent (generalization)

This operator uses a clause based revision point, which includes a list of the positive instances that flagged failure points in the clause. It tries to find antecedents that can be deleted to allow the instances to be proven, without allowing proofs of any negative instances. If necessary, the original clause will be generalized in several different ways. For example, suppose we "like" square blocks on tables and blocks on red tables, and we are given the overly specific rule

```
likeable(Block, Table) :-
        on(Block, Table), color(Table, red), shape(Block, square).
```

From this rule, delete-antecedent develops two generalizations to cover the positive instances:

```
likeable(Block, Table) :- on(Block, Table), color(Table, red).
likeable(Block, Table) :- on(Block, Table), shape(Block, square).
```

To do this, we generalize the original clause to cover as many positives as possible, without allowing proofs of any negatives. We then add the generalized clause to the theory. If there are more positives to be covered, we begin again with the original clause and repeat the process. We stop when all of the positive instances listed in the revision point are provable or we are unable to generalize the original clause to allow proof of any of the unprovable instances.

```
repeat
  if we can generalize the clause by delete-antecedent
    add generalized clause to the predicate
  else if we can generalize the clause by delete-multiple
    add generalized clause to the predicate
  else
    we are unable to generalize the clause
  end if
until all positives listed in this revision point are provable
   or we were unable to generalize the clause
if we have one or more generalizations
  replace the original clause with the generalizations
end if
```

Figure 13. The delete-antecedent operator.

We have two methods of generalization at our disposal. First, we try a hill-climbing approach to clause generalization. This method deletes one antecedent at a time, selecting each time the antecedent that allows the most unprovable positives to be proven. As with any hill-climbing approach, this is efficient but vulnerable to locality problems. If this approach fails, we use a more general method that can delete multiple antecedents simultaneously.

Hill-climbing antecedent deletion. This method of deleting antecedents is iterative. It tries deleting each antecedent in the specified clause, and notes two things: how many unprovable positives can be proven when the antecedent is deleted, and whether any negatives become provable as a result of its deletion. We select the antecedent that allows proof of the largest number of positives while not allowing any negatives to be proven. This antecedent is deleted, and the process repeats. We stop when there are no more antecedents whose deletion gains us anything.

```
repeat
  for each antecedent in the clause
    if deleting antecedent does not allow provable negatives
      count number of positives the deletion makes provable
    end if
  end for
  delete antecedent allowing the most provable positives
until we cannot delete any antecedent
if we didn't delete any antecedents
  fail
end if
```

Figure 14. Delete antecedent (hill-climbing).

This approach to deleting antecedents may fail—there may be no antecedent whose deletion allows positives to be proven but does not allow negatives to be proven. There are two principle causes of this. First, it may be that we need to add new discriminating antecedents to the clause after generalizing it. In this case, the add-rule operator is likely to propose a useful revision. Second, there may be many antecedents whose deletion does not appear to affect the provability of any instance—but we may be able to generalize the clause successfully by deleting several antecedents simultaneously. This local plateau problem is dealt with by our technique for deleting multiple antecedents.

Deleting multiple antecedents. Our second method of generalizing a clause deletes multiple antecedents. This method is more computationally expensive, since it may have to try deleting many different combinations of antecedents before finding one that is useful. However, it is a highly effective operator, capable of solving general "m of n" problems. For example, suppose that we are interested in blue, red, or green blocks on a table. The user may know the important features, but not the correct combinations, and provide the initial theory

```
likeable(Block, Table) :- on(Block, Table),
        color(Block, blue), color(Block, green), color(Block, red).
```

This theory is so overspecialized that no positives are provable (since a block can only be one color). Further, deletion of any single antecedent will not allow any positives to be proven. However, the algorithm for deleting multiple antecedents will correctly generate each of the three clauses

```
likeable(Block, Table) :- on(Block, Table), color(Block, red).
likeable(Block, Table) :- on(Block, Table), color(Block, green).
likeable(Block, Table) :- on(Block, Table), color(Block, blue).
```

To generalize a clause, we first collect all antecedents whose deletion does allow any negative instance to be proven. Note that none of these deletions will allow positive instances to be proven, or else the hill-climbing approach to antecedent deletion would have found them. We exhaustively generate combinations of these antecedents, looking for a combination whose deletion allows proof of one or more positives but no negatives.

```
delete-multiple
collect all antecedents whose deletion
  does not allow negatives to be proven
repeat
  repeat
    delete an antecedent
    if negatives are provable
      prune this branch of the search space
    end if
  until no antecedents left in the set
until one or more positives are provable
   or we have exhausted the search space
if no positives became provable
  fail
end if
```

Figure 15. Delete antecedent (delete-multiple).

We build our combination of deletions one antecedent at a time, working left-to-right through the clause. When we delete an antecedent, we check to see if any negatives have become provable. This allows us to substantially prune the search space, as, if negatives have become provable, we discard not only this particular combination but all supersets of it. Note that we do not stop when positives have become provable--we delete as many antecedents as we can, in an attempt to cover as many positives as possible.

5.3.3 Add antecedent (specialization)

If negative examples are provable, the theory must be specialized. The delete-rule operator is a drastic way to do this. A gentler approach is to specialize a clause by adding antecedents to discriminate between positive and negative instances. In order to retain proofs of all positive instances, we may need to create several different specializations of the original rule. For example, if we "like" red or green blocks on tables, then the rule

```
likeable(Block, Table) :- on(Block, Table).
```

is overly general. However, we can specialize it in two ways to produce the correct theory:

```
likeable(Block, Table) :- on(Block, Table), color(Block, red).
likeable(Block, Table) :- on(Block, Table), color(Block, green).
```

We have two methods of adding antecedents to a clause: a simple hill-climbing method (similar to the hill-climbing method we used to delete antecedents) and a method we call *relational pathfinding* [Richards and Mooney, 1992]. We always use both methods to specialize the original clause, and we select the better of the two specializations. If the selected specialization covers all positives, we are finished. Otherwise, we repeat the process, producing other specializations of the original clause.

```
repeat
  specialize original clause by hill-climbing
  specialize original clause by relational pathfinding
  choose the clause covering the most positives
  if both cover the same number of positives
    choose the smaller clause
  end if
  add the chosen clause to the proposed revision
until all positives covered by the original clause are covered
   or until we are unable to generate a specialization
if we produced one or more specialized clauses
  replace the original clause with the specializations
end if
```

Figure 16. The add-antecedent operator.

Since our two methods of adding antecedents have very different capabilities, they are mutually recursive. This means that one method can perform part of a specialization and the other can finish it. For example, consider the problem of learning a rule for an uncle who is a blood relative. Relational pathfinding will readily generate the clause

```
uncle(X, Y) :- parent(V, X), parent(V, W), parent(W, Y).
```

This rule will be a good discriminator between positive and negative instances, but it is still overly general. Hence, relational pathfinding will call the hill-climbing method to provide the final two antecedents:

```
uncle(X, Y) :-
        parent(V, X), parent(V, W), parent(W, Y), X \= W, gender(X, male).
```

Hill-climbing antecedent addition. The hill-climbing method begins by generating all variablizations of all antecedents that could be added to the current rule. The language bias determines which classes of antecedents are considered, and clearly invalid or redundant antecedents are not generated (for example, relational antecedents must contain at least one variable that appears elsewhere in the clause). The collected antecedents are scored on their ability to eliminate proofs of negative instances while retaining the provability of positive instances. We add the best antecedent, and iterate. The process stops when we have eliminated all negative instances or when no antecedent has a positive score.

```
repeat
  create set of possible antecedents
  score antecedents on their ability to concentrate
positives
  add the best antecedent
until all negatives are eliminated
   or no antecedent has a positive score
if we specialized the rule, but negatives are still provable
  call relational pathfinding with the specialized clause
end if
```

Figure 17. Add antecedent (hill-climbing).

The scoring function is similar to the one used by FOIL [Quinlan, 1990]. However, FOIL counts the number of proofs of instances, whereas FORTE counts the number of instances (ignoring the fact that one instance may be provable in several different ways). We believe that this provides a more realistic measure of the benefit of an antecedent. The scoring function measures the amount of information an antecedent conveys about the instances. To develop this measure, we determine how much information we need to correctly partition the instances both with and without the antecedent. The difference between these amounts is the information gain of the antecedent.

In order to correctly partition an arbitrary set, we can resort to appending a flag to each positive instance. The amount of information this requires is one bit per positive instance. However, if we have a disproportionate number of positives, we may be able to find a more efficient approach. Information theory gives us the theoretical limit of this. The minimum amount of information we must have for each positive instance is:

$$information = -\log_2(\, pos \,/\, (pos + neg)\,) \text{ bits}$$

where *pos* and *neg* are the numbers of positive and negative instances in the set. If we have equal numbers of positive and negative instances, this gives us the expected result

$$information = -\log_2(\, 1/2\,) = 1 \text{ bit}$$

per positive instance. If we have twice as many positives as negatives, we need only

$$information = -\log 2(\, 2/3\,) = 0.585 \text{ bits}$$

per positive instance. On the other hand, if we have twice as many negatives as positives, we need

$$information = -\log 2(\, 1/3\,) = 1.585 \text{ bits}$$

per positive instance. Note that this is an asymmetrical function, since we only measure the amount of information required to identify the positive instances. In theory, we might equally well partition the set by identifying the negative instances. However, we are writing rules to allow us to prove positives; hence, antecedents that help us concentrate negatives would only be useful if we were to introduce negation into our rules.

As an example, suppose we begin with a set containing eight positives and eight negatives. To partition this set, we need 1.0 bits of information for each positive instance. If we have an antecedent that makes four of the negatives unprovable, leaving us with eight positives and only four negatives, we need only 0.585 bits per positive instance. The information gain of the antecedent is

$$gain = before - after = 1.0 - 0.585 = 0.415 \text{ bits}$$

per positive instance. Hence, the total information gain for this antecedent is

$$ante\_score = gain * pos = 0.415 * 8 = 3.32$$

Thus, our final score for this antecedent is 3.32.

It is worth noting that this asymmetrical scoring heuristic allows the algorithm to terminate when antecedents that eliminate negative instances are still available. For example, if we begin with eight positive and eight negative instances as above, and the best antecedent available leaves three positives and five negatives still provable, this antecedent will have a score of -1.25. Since it has a negative score, it will not be selected. In this case, the algorithm would stop, retaining whatever antecedents had been added up to this point.

Relational pathfinding. Relational pathfinding is a method of antecedent addition designed to escape local maxima and local plateaus. The idea of pathfinding in a relational domain is to view the domain as a (possibly infinite) graph of constants linked by the relations that hold between the constants. For example, a portion of the data in Hinton's family domain

[Hinton, 1986] is shown in Figure 18. Relational pathfinding is particularly easy to visualize in this domain, since all relations are binary.

We can see an example of the local plateau problem in this domain by trying to define the grandparent relation using only one positive and one negative instance:

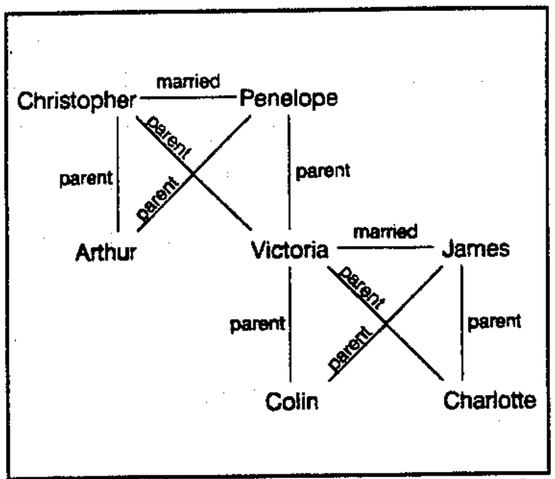

Figure 18. A family, showing people as nodes and relations as edges.

+ grandparent(Christopher, Colin)
- grandparent(Christopher, Arthur)

There is no single antecedent that will discriminate between these instances. Both Colin and Arthur have parents, neither has children, and neither is married. In order to create a correct theory, we must simultaneously add both of the required parent literals, i.e.,

grandparent(X, Y) :- parent(X, Z), parent(Z, Y).

Relational pathfinding is based on the assumption that, in most relational domains, important concepts will be represented by a small number of fixed paths among the constants defining a positive instance. For example, in this case, the grandparent relation is defined by a single fixed path consisting of two parent relations.

Relational pathfinding can be used anytime a clause needs to be specialized and does not have relational paths joining all of its variables. If, after pathfinding, the rule is still too general, we do further specialization using the hill-climbing technique. This arises, for example, when a rule requires non-relational antecedents.

Relational pathfinding finds paths by successive expansion around the nodes associated with the constants in a positive example, in a manner reminiscent of Quillian's spreading activation [Quillian, 1968]. We arbitrarily choose a positive instance and use it to instantiate the initial rule. The constants in the instantiated rule are nodes in the domain graph, possibly connected by antecedents in the rule. We then identify isolated subgraphs among these constants; if the initial rule contains no antecedents, then each constant forms a singular subgraph.

We view a subgraph as a nexus from which we explore the surrounding portion of the domain graph. Each exploration that leads to a new node in the domain graph is a path, and the value of the node it has reached is the path's end-value. Initially, each constant in a sub-graph is the end-value of a path of length zero.

Taking each subgraph in turn, we find all new constants that can be reached by extending any path with any defined relation. These constants form the new set of path end-values for the subgraph. We check this set against the sets of end-values for all other subgraphs, looking for an intersection. If we do not find an intersection, we expand the next node. This process continues until we either find an intersection or exceed a preset resource bound.

When we find an intersection, we add the relations in the intersecting paths to the original instantiated rule. If the new relations have introduced new constants that appear only once, we complete the rule by adding relations that hold between these singletons and other constants in the rule.

If we are unable to use all such singletons, the rule is rejected. Finally, we replace all constants with unique variables to produce the final, specialized theory clause. If we simultaneously discover several intersections, we develop clauses for all of them and choose the one that provides the best accuracy on the training set.

While the pathfinding algorithm potentially amounts to exhaustive exponential search, it is generally successful for two reasons. First, by searching from all nodes simultaneously, we greatly reduce the total number of paths explored before we reach an intersection. Second, most meaningful relations are defined by short paths, which inherently limits the depth of search. However, a practical implementation of this method includes a resource bound.

As an example, suppose we want to learn the relationship uncle, given an initially empty rule and the positive instance uncle(Arthur, Charlotte). The process is illustrated in Figure 19. We begin by exploring paths from the node labelled Arthur, which leads us to the new nodes Christopher and Penelope. We then expand from the node labelled Charlotte, leading to the nodes Victoria and James. At this point we still do not have an intersection, so we lengthen all paths originating from node Arthur. We eliminate any end-values that we have already used (and which, therefore, do not give us an intersection). This leaves us

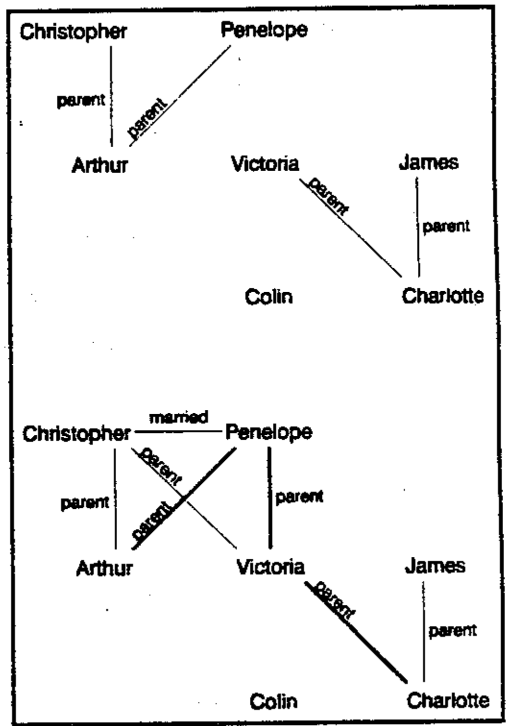

Figure 19. Finding the relational path for uncle.

with only one path end-value: Victoria. Since Victoria is also an end-value of one of the paths originating from Charlotte, we recognize an intersection.

There are two paths leading from Arthur to Victoria, but in this case they are identical (merely leading through different grandparents). If we had found several paths, we would select the one providing the best overall accuracy. The final path in this example is

uncle(X, Y) :- parent(Z, X), parent(Z, W), parent(W, Y)

The literal male(X), which is required to complete this rule, is not a relation and is therefore added by hill-climbing antecedent addition.

5.3.4 Add rule (generalization)

Add rule is a clause-based generalization operator that develops one or more new versions of an existing rule, while leaving the original rule in the theory. Its objective is to create a new rule that allows proof of positive instances that identified the original rule as a failure point. Building this new rule is a three-step process. First, we make a copy of the original rule. Second, we deliberately overgeneralize it to allow proof of the positives that failed on the original rule. The goal is to reduce the rule to a core of antecedents that are essential to keep negatives from being provable, while not interfering with proofs of positives. To this end, we delete all antecedents whose deletion does not allow any negatives to be proven, and we also delete all antecedents whose deletion allows one or more positives to be proven (even if doing so allows negatives to be proven). Third, we use the

```
copy rule to be generalized
overgeneralize the rule to allow positives to be proven
call add-antecedent to eliminate provable negatives
if add-antecedent produced one or more specializations
  add the specializations to the theory
end if
```

Figure 20. The add-rule operator.

add-antecedent operator to specialize the overgeneralized clause to eliminate any newly provable negatives.

As an example, suppose we "like" red, green, and blue blocks on a table. Suppose we begin with the correct but insufficient clause

```
likeable(Block, Table) :- on(Block, Table), color(Block, red).
```

With this theory, all positive instances with green or blue blocks are unprovable. Add-rule copies this rule, and overgeneralizes it by deleting the antecedent for color. The result is rule

```
likeable(Block, Table) :- on(Block, Table).
```

This rule allows all positives to be proven, but also many negatives (for instance, yellow blocks on tables). Hence, add-rule calls add-antecedent to specialize the rule and eliminate the negative instances. Add-antecedent creates the two specializations

```
likeable(Block, Table) :- on(Block, Table), color(Block, green).
likeable(Block, Table) :- on(Block, Table), color(Block, blue).
```

These two new rules are added to the theory, and the original rule is left intact.

5.3.5 Identification (generalization)

Identification is a predicate-based operator. It seeks to construct a new clause to generalize the definition of an antecedent that caused one or more proofs of positive instances to fail. Rather than developing a new clause from scratch, it performs an inverse resolution step using two existing rules in the domain theory. In chapter 3 we saw a simple propositional example:

$$\frac{\begin{array}{l} a \leftarrow b, x \\ a \leftarrow b, c, d \end{array}}{\begin{array}{l} a \leftarrow b, x \\ x \leftarrow c, d \end{array}}$$

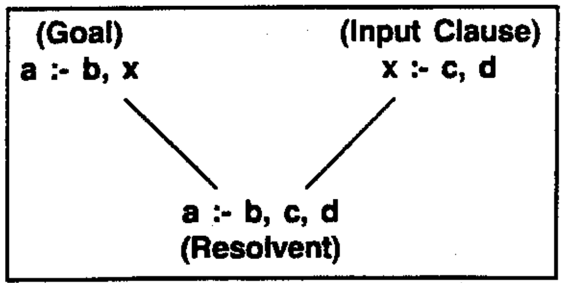

Figure 21. A resolution step.

This has no effect on the deductive closure of these rules alone, but we have generalized the theory by adding a new rule for predicate x. This process essentially reverses one resolution step; from a goal and a resolvent we have derived an input clause.

Unification makes this a more complex process in first-order domains. Suppose our initial theory of family relationships includes the following rules, where aunt_uncle is intended to be a general rule for identifying aunts and uncles without regard to sex.

```
uncle(A, B) :- gender(A, male), aunt_uncle(A, B).
uncle(C, D) :- gender(C, male), sibling(C, E), parent(E, D).
aunt_uncle(A, B) :- married(A, C), sibling(C, D), parent(D, B).
aunt(A, B) :- gender(A, female), aunt_uncle(A, B).
```

When we are presented with an instance of an aunt who is a blood relative, this instance will not be provable. One of the failure points is the call to aunt_uncle clause. Identification looks for ways to provide an alternate definition of this rule.

Our algorithm works as follows. First, we can only develop a revision if the predicate identified is one defined in the theory (as opposed to, say, a predicate in the fundamental domain theory). We then look through all predicates in the theory, and within each predicate, we look for pairs of clauses that meet our criteria (described below). Note that there may be several suitable pairs of clauses in the theory. Identification will propose one revision for each such pair. In our example, we will derive a clause from the two rules for uncle/2.

```
if the predicate identified in the revision point is a theory predicate
  for all predicates in the theory
    select two clauses
    heuristic checks, e.g., goal must call revision-point predicate
    unify consequent of goal and consequent of resolvent
    find matching antecedents in goal and resolvent
    for each antecedent
      goal antecedent more general than resolvent antecedent
      unify the antecedents
    end for
    if goal has exactly one unmatched term, and it is the predicate we want to generalize
      unmatched term in goal is consequent of input clause
      unmatched terms in resolvent are antecedents of input clause
      if input clause meets language bias
        propose replacing resolvent with input clause
      end if
    end if
  end for
end if
```

Figure 22. The identification operator.

Given a pair of clauses, we choose one to be the goal and the other to be the resolvent. We unify their consequents, requiring the consequent of the goal to be more general than the consequent of the resolvent. We then scan their bodies, matching as many antecedents as possible. We unify each matched pair, again requiring literals from the goal to be more general than literals from the resolvent. The generality requirement prevents us from inadvertently reducing the deductive closure of the two rules. When we have finished the matching process, we must have exactly one unmatched antecedent left in the goal, and this must be a call to the predicate we want to generalize. There must also be some number of unmatched antecedents left in the resolvent. The unmatched antecedent in the goal becomes the consequent of the input clause, and the unmatched antecedents in the resolvent become the antecedents. We then check the derived clause to ensure that it meets the language bias. In our example, we derive the clause

```
aunt_uncle(A, B) :- sibling(A, E), parent(E, B).
```

Note that, if we delete the second clause of uncle/2 and add this clause to the theory, we will still be able to prove all instances of uncle. However, we have generalized the definition of aunt/2. Our revised theory is:

```
uncle(A, B) :- gender(A, male), aunt_uncle(A, B).
aunt_uncle(A, B) :- sibling(A, E), parent(E, B).
aunt_uncle(A, B) :- married(A, C), sibling(C, D), parent(D, B).
aunt(A, B) :- gender(A, female), aunt_uncle(A, B).
```

5.3.6 Absorption (generalization)

Absorption is the complement of identification. It uses antecedent-based revision points. Rather than constructing a new clause for the predicate corresponding with the failing antecedent, absorption looks for a predicate that subsumes the failing antecedent (and possibly other antecedents in the rule) and which may already have alternate clauses that will allow the positive instances to be proven. The propositional example in chapter 3 was

$$\begin{array}{l} a \leftarrow b,\ c,\ d \\ x \leftarrow c,\ d \\ \hline a \leftarrow b,\ x \\ x \leftarrow c,\ d \end{array}$$

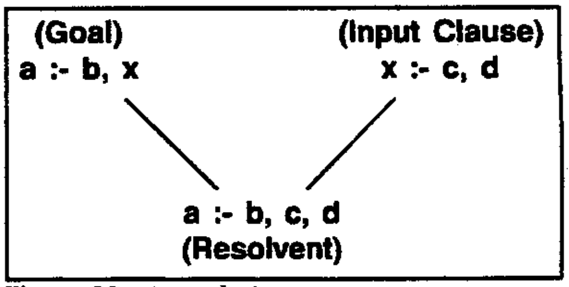

Figure 23. A resolution step.

This generalized the theory by allowing alternate definitions of x to be used when proving a. As with identification, absorption reverses a single resolution step. In this case, we are deriving a goal from a resolvent and an input clause.

As with identification, in first-order logic we have to choose variable substitutions in a way that ensures we do not inadvertently specialize the theory. Suppose our theory includes the rules

```
uncle(A, B) :- gender(A, male), sibling(A, C), parent(C, B).
aunt_uncle(D, F) :- sibling(D, E), parent(E, F).
aunt_uncle(A, B) :- married(A, C), sibling(C, D), parent(D, B).
```

When we are presented with an instance of an uncle who is not a blood relative, we will not be able to prove it using this theory. We will have a failure point either at sibling/2 or parent/2. Absorption looks for other rules in the theory that have antecedents similar to the one in the failure point, in hopes of finding a rule that will subsume some of the antecedents in the current clause. If there are several candidate rules, absorption will propose several revisions. In our example, absorption finds the first clause of aunt_uncle/2 to be a possibility.

```
clause identified in the revision point is the resolvent
antecedent identified in the revision point is the target antecedent
for each clause in the theory containing a more general version of the target antecedent
  the identified clause is our input clause
  find matching antecedents in goal and input clause
  for each antecedent
    input clause antecedent more general than resolvent antecedent
    unify the antecedents
  end for
  all input clause must be matched
  consequent of resolvent is consequent of goal
  unmatched terms in resolvent plus call to input clause are antecedents of goal
  verify goal (e.g., cannot be recursive unless language bias allows recursion)
end for
```

Figure 24. The absorption operator.

In order to develop a revision, we consider the original uncle clause to be the resolvent, and the aunt_uncle rule to be the input clause. We then match antecedents between the two clauses. In order to be successful, we must be able to match every antecedent in the input clause with a corresponding antecedent in the resolvent. Moreover, each antecedent in the input clause must be more general than its counterpart. If we can match all antecedents in the input clause, we replace the matched antecedents in the resolvent with a call to the input clause, using the variable substitutions automatically derived through unification. If there is more than one way to match the antecedents, absorption will propose one revision for each mapping. In this case, we have developed the new clause

```
uncle(A, B) :- gender(A, male), aunt_uncle(A, B).
```

Hence, the revised theory would be

```
uncle(A, B) :- gender(A, male), aunt_uncle(A, B).
aunt_uncle(D, F) :- sibling(D, E), parent(E, F).
aunt_uncle(A, B) :- married(A, C), sibling(C, D), parent(D, B).
```

5.3.7 Identification (compaction)

Identification can often reduce the size of the theory. Hence, we try the identification algorithm on all suitable pairs of clauses in the theory. If it is able to reduce the size of the theory without reducing accuracy, it proposes the change as a revision. This operator is otherwise identical to the identification operator used for generalization.

5.3.8 Absorption (compaction)

Similarly, absorption can reduce the size of a theory. We try absorption on all suitable pairs of clauses, and propose a revision whenever absorption identifies a change that reduces the theory size without reducing theory accuracy. This operator is otherwise identical to the absorption operator used for generalization.

5.4 COMPUTATIONAL COMPLEXITY

This section presents a brief look at of FORTE's computational complexity, along with empirical verification of that complexity. As one would expect, FORTE's complexity is exponential in the size of the input theory and in the arity of the theory predicates. For example, when FORTE is considering new antecedents for addition to a rule, the number of permutations of arguments to a predicate is an exponential function of the predicate's arity. However, FORTE's complexity for a given learning problem, where those items are fixed, is quadratic in the size of the training set. For sufficiently large training sets, FORTE's complexity is approximately linear.

A simplified explanation of FORTE's theory revision process reveals the reason for the quadratic bound. During the theory revision process, FORTE will have to make at most one revision for each improperly classified instance; this is one factor of *n*. Each revision must be developed and evaluated on the entire training set (and the amount of work to do so is determined exclusively by the domain and initial theory); this is the second factor of *n*. Hence, quadratic complexity[16].

In practice, however, there is a threshold on the size of the training set. Once FORTE has enough instances to develop a complete and correct theory, the number of revisions needed to develop this theory does not increase further with larger training sets. Thus, any additional increases in training set size produce only a linear increase in execution time.

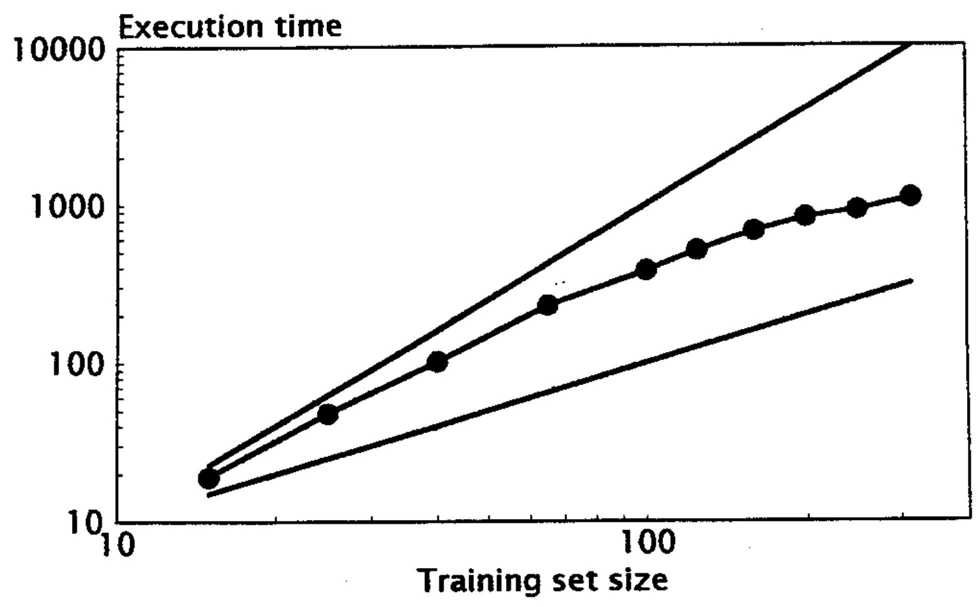

Figure 25. An empirical measurement of FORTE's computational complexity. The upper and lower lines are quadratic and linear bounds, respectively. Each point was averaged over 30 trials.

[16]This assumes a constant proof time. Empirical results bear this out as a valid approximation in the average case.

We ran an experiment to verify this quadratic/linear performance. In Figure 25 we see a graph of FORTE's execution time while performing inductive learning in Hinton's family domain. To produce this graph, we recorded FORTE's average execution time for a variety of training set sizes. The graph is on a log-log scale, which means that polynomials show up as lines, and the slope of a line is proportional to the degree of the polynomial. The lower line is a linear complexity bound, and the upper line is quadratic. FORTE's execution time falls between these two bounds. With small training sets the slope of FORTE's line nearly parallels the quadratic bound, as we expect. As the number of instances passes the number required for FORTE to produce a complete and correct theory, the slope of FORTE's line decreases, approaching that of the linear bound.

Chapter 6
RESULTS IN STANDARD LEARNING DOMAINS

In this chapter we look at the results of using FORTE in standard first-order machine-learning domains, and show that using theory revision with an approximately correct theory is much more effective than pure induction in these domains. We apply FORTE to two standard first-order machine learning problems: family relationships and king-rook-king illegality. These domains were chosen to be substantially different from one another; the family relationships domain is highly relational, whereas the king-rook-king domain is nearly nonrelational, and could be solved as a propositional problem.

The learning curves in this chapter represent the accuracy of FORTE's revised theories on unseen data. Since FORTE's hill-climbing techniques make it vulnerable to local maxima, another important aspect of learning is the accuracy of revised theories on the training data used for learning. In practice, FORTE has very little trouble with local maxima. During the first 1300 test runs used to generate the learning curves in the chapter, FORTE was caught in local maxima nine times (0.69%). In all nine cases, the accuracy of the revised theory on the training data was greater than 98%.

6.1 FAMILY DOMAIN

Many of the illustrative examples in Chapters 3 and 4 made use of Hinton's family domain, which appeared in [Hinton, 1986] and [Quinlan, 1990] as a test domain. While Hinton's data is suitable for small demonstrations, it includes a great deal of artificial structure (for example, all married couples have two children, one boy and one girl). In order to provide a more realistic test in a similar domain, we created a large, diverse family composed of 86 people across 5 generations. This domain uses the same twelve concepts as Hinton's data: husband, wife, mother, father, sister, brother, son, daughter, aunt, uncle, niece, and nephew. The first subsection below discusses the training data, the fundamental domain theory, and the test methodology. Subsequent sections provide the test results in this domain.

6.1.1 Methodology

The family data set includes 744 positive instances and 1488 randomly generated negative instances. Each test run uses an independent, randomly selected subset of these instances as the training set, with the remaining instances used as the test set. The family data provides the gender of each person, all marriages, and all parent-child relationships.

The revision verifier is empty. The fundamental domain theory provides a commutative definition for married/2. The relation-tuning parameter in the language bias was set to highly-relational.

```
wife(X, Y) :- gender(X, female), married(X, Y).
husband(X, Y) :- gender(X, male), married(X, Y).
mother(X, Y) :- gender(X, female), parent(X, Y).
father(X, Y) :- gender(X, male), parent(X, Y).
daughter(X, Y) :- gender(X, female), parent(Y, X).
son(X, Y) :- gender(X, male), parent(Y, X).
sister(X, Y) :- gender(X, female), sibling(X,Y).
brother(X, Y) :- gender(X, male), sibling(X,Y).
uncle(Z, Y) :- gender(X, male), au(X, Y).
aunt(Z, Y) :- gender(X, female), au(X, Y).
niece(X, Y) :- gender(X, female), au(Y, X).
nephew(X, Y) :- gender(X, male), au(Y, X).
au(X, Y) :- sibling(X, B), parent(B, Y).
au(X, Y) :- married(X, A), sibling(A, C), parent(C, Y).
sibling(X, Y) :- parent(A, X), parent(A, Y), X \= Y.
```

Figure 26. Correct family theory.

The theory revision tests used randomly corrupted versions of the correct theory shown in Figure 26. The number of errors introduced varied, and is specified with the test results below. Six possible errors could be introduced:

— Delete rule

— Add rule (1-3 antecedents)

— Delete antecedent

— Add antecedent

— Change antecedent (delete-ante plus add-ante)

— Change variable

When adding a new antecedent, there was a 50% chance that the antecedent used would be taken from elsewhere in the theory, and a 50% chance that it would be newly constructed. When changing a variable, there was a 50% chance that it would be changed to a variable appearing elsewhere in the same rule and a 50% chance that it would be changed to a new variable.

6.1.2 Theory Revision

One premise of theory revision is that it is easier to revise a theory that is mostly correct than it is to induce a new theory from scratch. In order to verify this premise, we generated five corrupted theories, containing an average of 3.6 errors each. Their average initial accuracy was 91.65%. Figure 27 shows the revision learning curve, which is averaged across four runs on each of the five theories, and an induction curve averaged over 20 independent trials (i.e., FORTE learning with no initial theory)[17]. The training set for each test run was independently generated—there was no relationship between the training sets used for revision and induction, nor were larger training sets supersets of smaller ones.

The difference between the curves at all training-set sizes is statistically significant[18] ($p > 0.99$). These results show that beginning with an approximate domain theory not only provides an initial boost in accuracy, but also

[17]A sample run of FORTE in this domain appears in Appendix B.

[18]Statistical significance was established using the standard non-paired t-test. Probabilities shown indicate the statistical likelihood that the difference is significant. Hence, ($p > 0.99$) indicates that there is less than a 1% chance that the difference between any pair of points in the learning curves was purely due to chance.

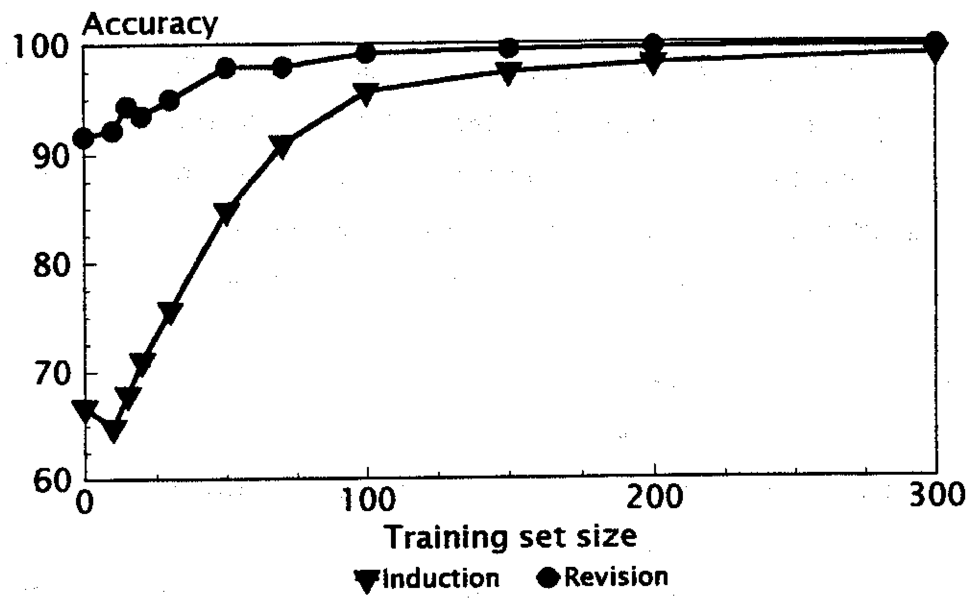

Figure 27. Induction vs. theory revision in the family domain.

that this advantage is maintained as the training set size increases.

Another performance issue in theory revision is how a system responds to increasing degradation of the initial theory. A good system will degrade gracefully as the accuracy of the input theory decreases. To illustrate this characteristic of FORTE we created five series of increasingly corrupted theories. Each series contains four theories, which, containing from two to eight errors each. We fixed the training set size, and ran FORTE four times on each corrupted theory in each series, and then averaged the results for each level of corruption (i.e., we averaged together the 20 runs on theories containing two errors, the 20 runs on theories containing four errors, and so forth). We repeated this experiment for training set sizes of 50 and 100 instances. The results appear in Figure 28.

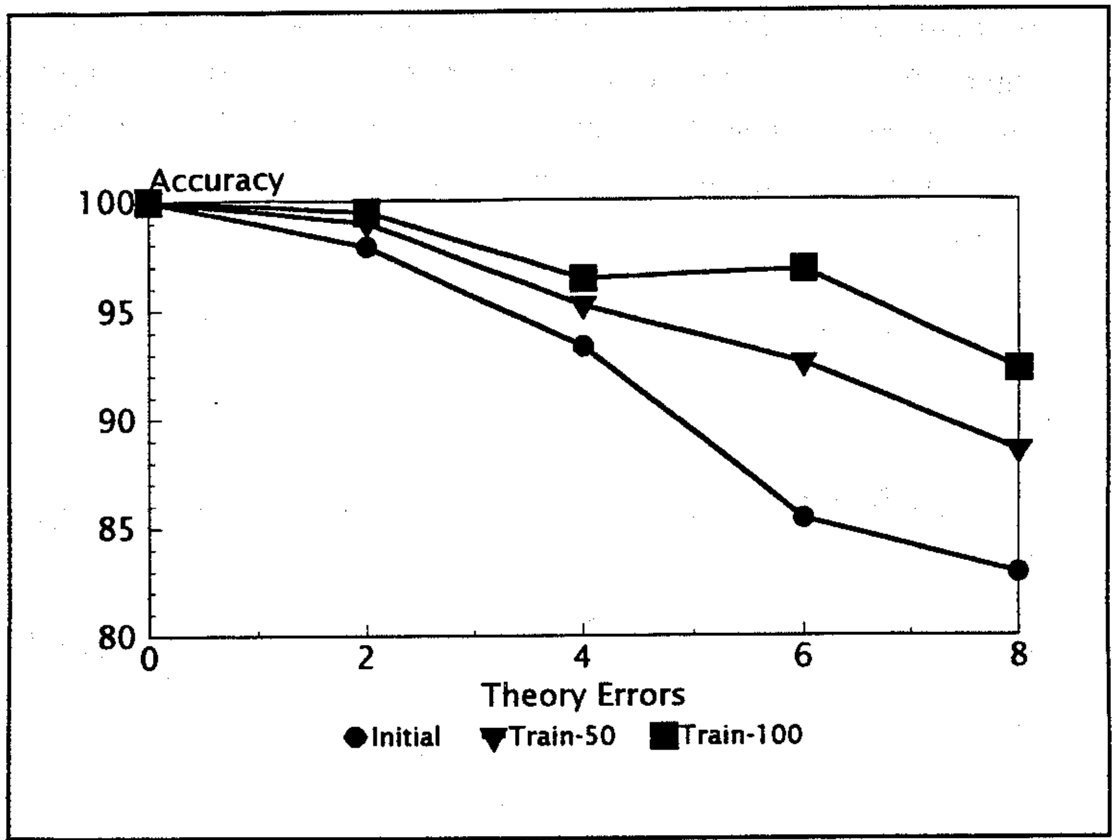

Figure 28. Revision accuracy for fixed training-set sizes, with increasingly corrupted theories.

The lowest curve shows the accuracy of the initial corrupted theories. The center curve shows the accuracy of FORTE's theories when it is given 50 training instances. The highest curve shows FORTE's accuracy when it is given 100 training instances. As expected, increasingly inaccurate initial theories do lower the accuracy of FORTE's revised theories for a given training set size. However, the degradation is gradual, and FORTE's output theories are always significantly better than the input theories ($p > 0.99$).

6.1.3 Advantage of Relational Pathfinding

The family domain is a prototypical first-order domain, in that it depends heavily on relations such as `parent(X, Y)` and `married(X, Y)` that cannot easily be translated into a propositional representation. Much of FORTE's performance in highly relational domains of this sort comes from relational pathfinding [Richards and Mooney, 1992]. To demonstrate this, Figure 29 shows FORTE performing inductive learning both with and without relational pathfinding. These curves are averaged over 20 runs for each data point. The difference between them is statistically significant ($p > 0.99$) at all points.

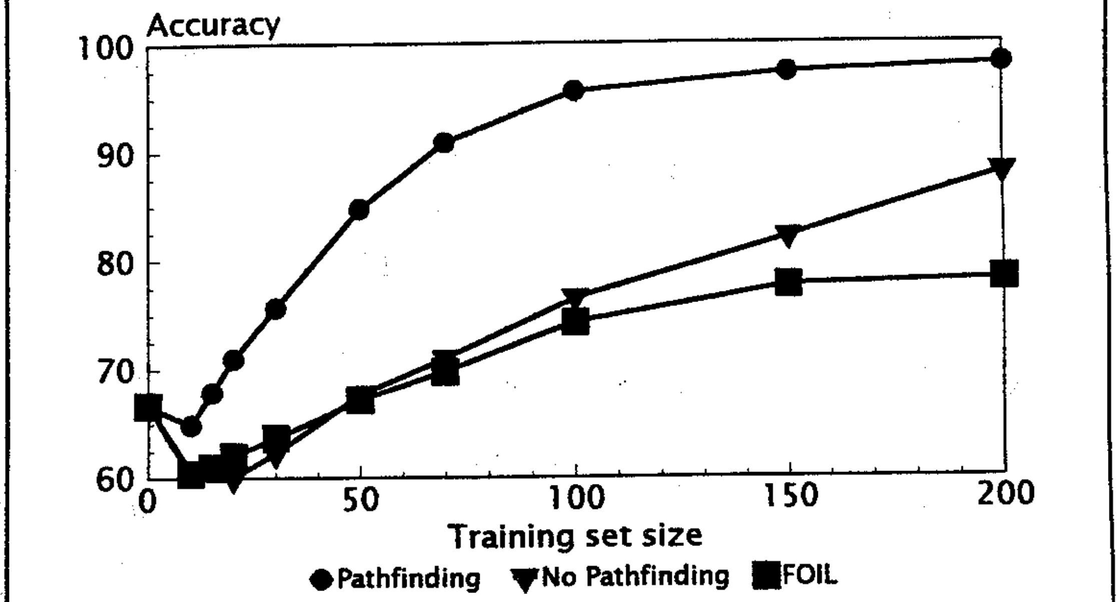

Figure 29. Comparison of FORTE (with and without relational pathfinding) and FOIL.

Figure 29 also includes a learning curve for FOIL[19] [Quinlan, 1990], averaged over 20 trials. We ran tests on FOIL using determinate literals [Quinlan, 1991], but the results were not significantly different from the basic FOIL curve shown; this is to be expected since, in this domain, few determinate literals are available. What is surprising is that FORTE, without using

[19]The FOIL experiments were run using an implementation of FOIL written by John Zelle.

relational pathfinding, produces significantly better results than FOIL for training set sizes 100 and higher. The primary algorithmic difference between FORTE (without relational pathfinding) and FOIL (without determinate literals) is that FOIL looks for all proofs of each instance whereas FORTE only cares whether or not an instance is provable. This leads to a different bias when selecting literals to add to clauses. In this case, that difference in bias favors FORTE.

6.2 KING-ROOK-KING ILLEGALITY

The king-rook-king illegality problem asks a learner to recognize illegal chess positions. The board contains a white rook and king, and a black king; white is to move. A position is illegal if two pieces occupy the same square or if black's king is in check. This domain is becoming a standard first-order test domain; it has been used in [Pazzani and Kibler, 1992], [Bain, 1991], [Cohen, 1991], and [Cohen, 1992].

In this domain we ran the tests of theory revision vs. induction and theory revision across increasingly corrupt initial theories. We did not run a test comparing FORTE's inductive performance with and without relational pathfinding, as relational pathfinding does not improve FORTE's performance in this domain.

6.2.1 Methodology

The KRK data set contains 2000 randomly generated instances, of which 684 are positive and 1316 are negative. Each test run uses an independent, randomly selected subset of these instances as the training set, with the remaining instances used as the test set. An instance in the KRK data provides the rank and file of each piece. There are no relational facts associated with the instances.

```
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, WRR, WRF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WRR, WRF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- adjacent_squares(WKR, WKF, BKR, BKF).
krk(WKR, WKF, Rank, WRF, Rank, BKF) :- line_attack(WKR, Rank, WKF, WRF, BKF).
krk(WKR, WKF, WRR, File, BKR, File) :- line_attack(WKF, File, WKR, WRR, BKR).

same_square(A, B, A, B).

adjacent_squares(Rank1, File1, Rank2, File2) :- adj(Rank1, Rank2), adj(File1, File2).
adjacent_squares(Rank1, File, Rank2, File) :- adj(Rank1, Rank2).
adjacent_squares(Rank, File1, Rank, File2) :- adj(File1, File2).

line_attack(WK, Others, A, B, C) :- not_equal(WK, Others).
line_attack(Same, Same, WK, WR, BK) :- less(WK, WR), less(WK, BK).
line_attack(Same, Same, WK, WR, BK) :- less(WR, WK), less(BK, WK).
```

Figure 30. Correct theory for the king-rook-king domain.

The revision verifier is empty. The fundamental domain theory provides definitions for rank and file adjacency, meaning that two numbers differ by exactly 1 (adj/2 and not_adj/2). It also includes definitions for less/2, not_less/2, equal/2, and not_equal/2. The relation-tuning parameter in the language bias was set to non-relational.

The theory revision tests used randomly corrupted versions of the correct theory shown in Figure 30. The number of errors introduced varied, and is specified with the test results below. Six possible errors could be introduced:

- — Delete rule
- — Add rule (1-3 antecedents)
- — Delete antecedent
- — Add antecedent
- — Change antecedent (delete-ante plus add-ante)
- — Change variable

When adding a new antecedent, there was a 50% chance that the antecedent used would be taken from elsewhere in the theory, and a 50% chance that it would be newly constructed. When changing a variable, there was a 50% chance that it would be changed to a variable appearing elsewhere in the same rule and a 50% chance that it would be changed to a new variable.

6.2.2 Theory Revision vs. Induction

Figure 31 compares inductive and theory revision learning curves in this domain. The corrupted theories were produced as described in Section 6.2.1; they contained an average of 3.2 errors, and had an average initial accuracy of 54.70%. As expected, beginning with an initial theory provides a dramatic boost both to the initial accuracy and to the steepness of the learning curve. This advantage is maintained for all training set sizes tested, and the difference in accuracy is statistically significant ($p > 0.99$).

We also tested FORTE's response to gradual degradation of the KRK theory. As in the family domain, we expect the accuracy of FORTE's revised theories to drop gradually as the accuracy of the initial theory decreases, for a fixed training set size. To illustrate this, we created five series of increasingly corrupted theories. Each series consists of four theories, containing from one to four errors each. We fixed the training set size, and ran FORTE four times on each corrupted theory in each series, and then averaged the results for each level of corruption (i.e., we averaged together the 20 runs on theories containing one error, the 20 runs on theories containing two errors, and so forth). We performed this experiment for training set sizes of 20 and 50 instances. The results appear in Figure 32.

The lowest curve shows the accuracy of the initial corrupted theories. The center curve shows the accuracy of FORTE's theories when it is given 20 training instances. The highest curve shows FORTE's accuracy when it is given

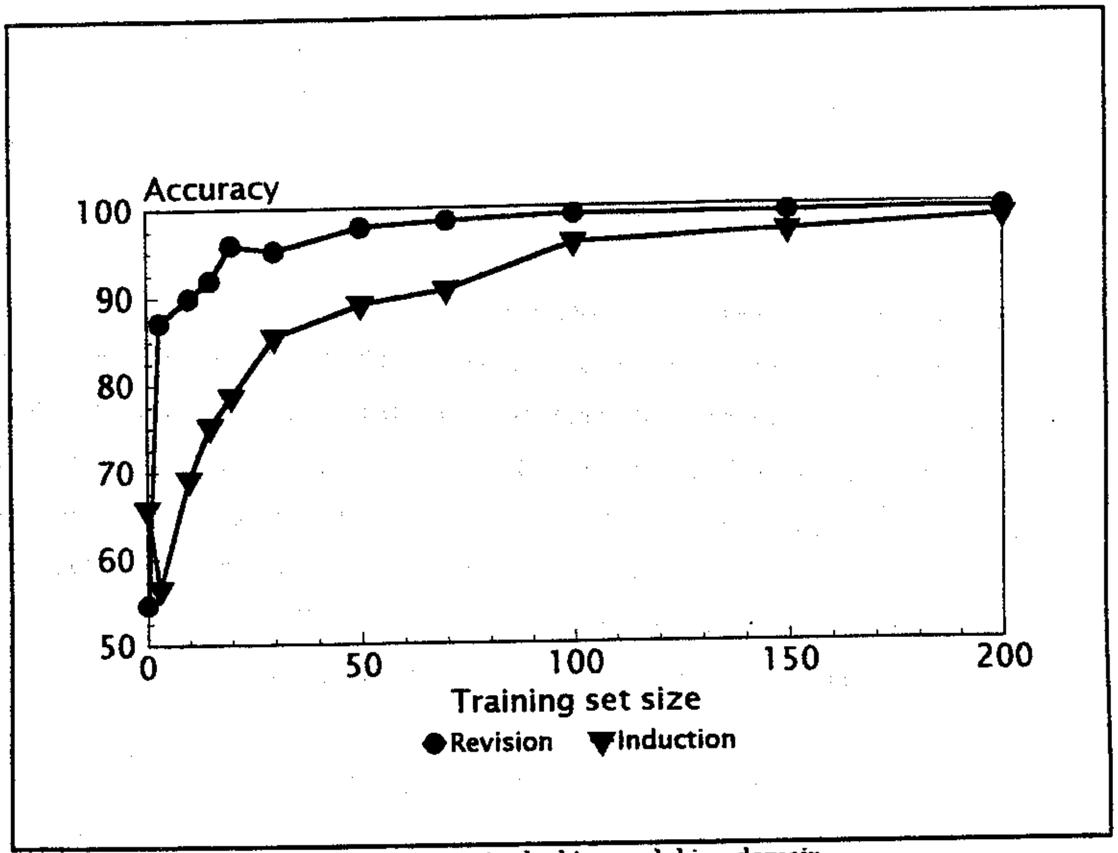

Figure 31. Induction vs. theory revision in the king-rook-king domain.

50 training instances. As expected, increasingly inaccurate initial theories do lower the accuracy of FORTE's revised theories for a given training set size. However, the degradation is gradual, and FORTE's output theories are always significantly better than the input theories ($p > 0.99$).

6.2.3 Comparison to Other Systems

Many researchers have tested their first-order systems on KRK data. Unfortunately, most such tests have provided only a single data point. This section summarizes the performance results of other systems and compares them to FORTE.

CIGOL. [Bain, 1991] presents results for two versions of CIGOL performing inductive learning on this data. Using a training set of 5000

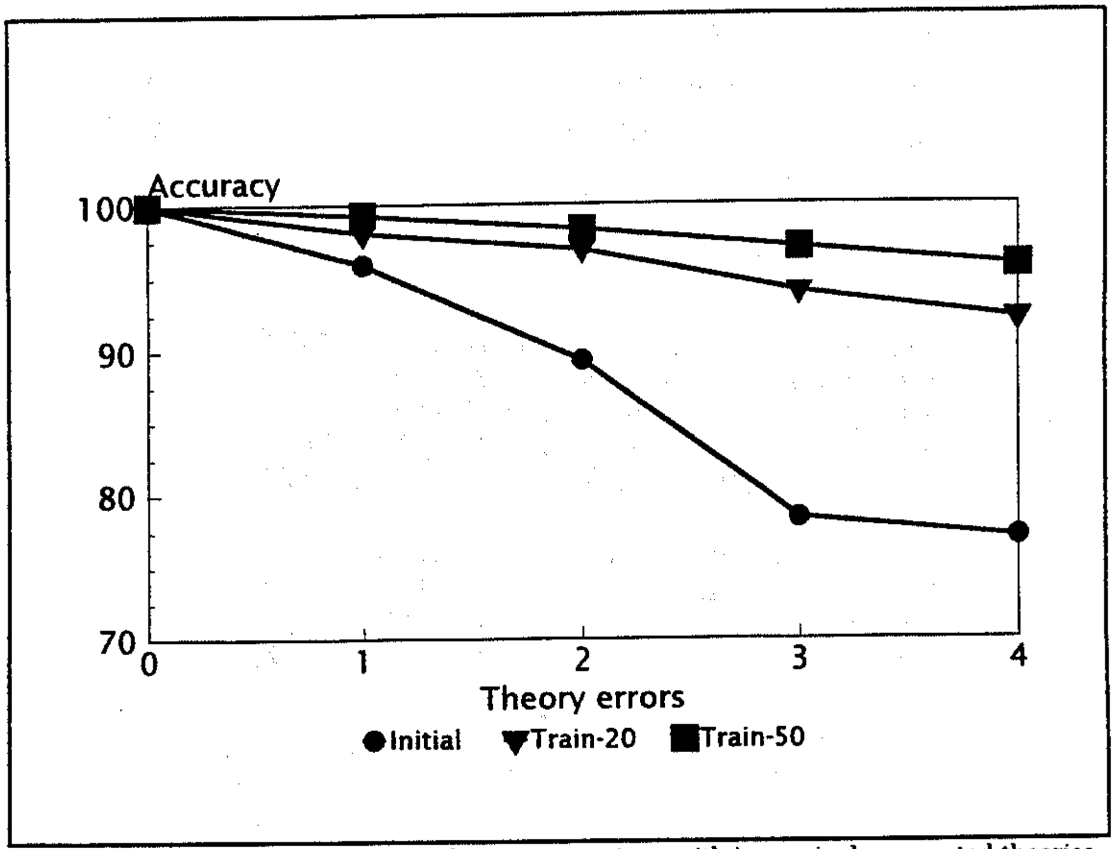

Figure 32. Revision accuracy for fixed training-set sizes, with increasingly corrupted theories.

instances, basic CIGOL never achieves an accuracy greater than 90%. However, with the introduction of what Bain calls closed-world specialization, CIGOL is able to achieve an accuracy of 100%. FORTE has not been run on a training set of that size; however, with a training set of 1000 instances it achieves an accuracy of 99.6%. The missing fraction is caused by the relatively rare circumstance of a black king that would be in check by the white rook, except for the interposition of the white king (this arises in fewer than 1 out of 200 instances). It seems likely that 5000 instances would be sufficient to allow Forte to learn this unusual condition.

FOCL. In [Cohen, 1991] and [Cohen, 1992], Cohen discusses the performance of FOCL on this data set. When FOCL learns inductively (i.e., equivalent to FOIL) using a training set of 100 instances, it develops theories that are 96.7% accurate. FORTE averages 95.8% accuracy on the same size training set. Here, the slight difference in bias that gave FORTE an advantage in the family domain instead gives an advantage to FOIL and FOCL. Cohen also presents FOCL results using an initial incorrect theory from [Pazzani and Kibler, 1992]. Using the initial theory

```
krk(A,B,C,D,E,F) :- same_square(A,B,C,D), adj(B,F).
krk(A,B,C,D,E,F) :- same_square(A,B,E,F).
krk(A,B,C,D,E,F) :- same_square(C,D,E,F).
krk(A,B,C,D,E,F) :- king_attacks_king(A,B,E,F).
krk(A,B,C,D,E,F) :- rook_attacks_king(A,B,C,D,E,F).

king_attacks_king(A,B,E,F) :- adj(A,E), adj(B,F).
king_attacks_king(A,B,E,F) :- adj(A,E), equal(B,F).
king_attacks_king(A,B,E,F) :- equal(A,E).
king_attacks_king(A,B,E,F) :- knight_move(A,B,E,F).

rook_attacks_king(A,B,C,D,E,F) :-
        equal(D,F), king_not_between_rank(A,B,C,D,E,F).

king_not_between_rank(A,B,C,D,E,F) :- not_equal(B, D).
king_not_between_rank(A,B,C,D,E,F) :- equal(B, D), not_between(C, A, E).

king_not_between_file(A,B,C,D,E,F) :- not_equal(A, C).
king_not_between_file(A,B,C,D,E,F) :- equal(A, C), not_between(D, B, F).
```

FOCL produces revised theories that average 97.9% accurate. FORTE, using the same initial theory, averages 95.6% accurate—essentially the same as it achieves for pure induction. The reason for this poor performance is that the corrupted theory contains two overly-general clauses in the same predicate. Since FORTE cannot correct two clauses simultaneously, it instead inductively specializes the next-higher-level-level predicate in the theory, thereby losing the information still available in the overly-general predicate. FORTE's average theory revision performance for training sets of this size is much higher, 99.1% on the theories in Section 6.2.2.

An interesting related problem is that of detecting only file attacks in the KRK scenario (i.e., when the black king is under attack along a file by the white rook). A correct theory for this domain is

```
file_attack(A,B,C,D,E,F) :-
        same_file(C,D,E,F), unblocked_king(A,B,C,D,E,F).
same_file(C,D,E,F) :- D = F.
unblocked_king(A,B,C,D,E,F) :- B \= D.
unblocked_king(A,B,C,D<E<F) :- not_between(C,A,E).
```

Cohen uses a non-uniform data set where the rook and black king are on the same file about two-thirds of the time. When the rule for same_file/4 is deleted from this theory, FOCL's accuracy (given a training set of 25 instances) averages 93.2%. Cohen presents a theory revision system, SIMPLE, that is only capable of adding missing rules, that achieves 100% accuracy on this theory. FOCL is unable to use large parts of the initial theory when a lower-level rule is missing, and must resort to pure inductive learning at the top level.

When this problem is given to FORTE, which can add lower-level rules to a theory, FORTE's average accuracy is 99.4%. This is significantly better than FOCL, but not as good as SIMPLE. While FORTE can add the missing lower-level rule, it can also correct the theory at the top level. Which correction it makes depends on the particular instances present in the training set.

6.2.4 Differently Structured Theories

One of the important capabilities of FORTE is its ability to modify rules at any level in a theory. In the corrupted theories of Section 6.2.2, many of the errors appeared in the predicates same-square, adjacent-squares, and line-attack. FORTE revised the errors in these predicates as easily as errors in the top-level predicate. To demonstrate that FORTE's revision process is equally effective at any level in a theory, we created a second correct theory for the KRK domain. This version, shown in Figure 33, consists solely of top-level clauses.

```
krk(Rank, File, Rank, File, BKR, BKF).
krk(Rank, File, WRR, WRF, Rank, File).
krk(WKR, WKF, Rank, File, Rank, File).

krk(Rank, WKF, WRR, WRF, Rank, BKF) :- adj(WKF, BKF).
krk(WKR, File, WRR, WRF, BKR, File) :- adj(WKR, BKR).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- adj(WKF, BKF), adj(WKR, BKR).

krk(Rank1, WKF, Rank, WRF, Rank, BKF) :- not_equal(Rank, Rank1).
krk(Rank, WKF, Rank, WRF, Rank, BKF) :- less(WKF, WRF), less(WKF, BKF).
krk(Rank, WKF, Rank, WRF, Rank, BKF) :- less(WRF, WKF), less(BKF, WKF).

krk(WKR, File1, WRR, File, BKR, File) :- not_equal(File, File1).
krk(WKR, File, WRR, File, BKR, File) :- less(WKR, WRR), less(WKR, BKR).
krk(WKR, File, WRR, File, BKR, File) :- less(WRR, WKR), less(BKR, WKR).
```

Figure 33. Correct single-level theory for king-rook-king illegality.

We then created five randomly corrupted version of this theory (with the same average of 3.2 errors per theory) and ran FORTE on them. Figure 32 compares the learning results on these theories with the results shown earlier using the multi-level theory. The average initial accuracies of the two sets of corrupted theories are very different (54.7% vs. 81.4%). This is a result of their different structures. When an over-generalization error is introduced into the flat theory, it can potentially affect all training instances. In the multi-level theory, the error may be in one of the lower-level predicates, in which case it can only affect a small portion of the training set.

In spite of the initial difference in accuracy, the two learning curves quickly approach each other. For a training set size of 20, the difference between the two curves is still statistically significant ($p > 0.99$). However, the revision accuracies continue to approach each other as the size of the training

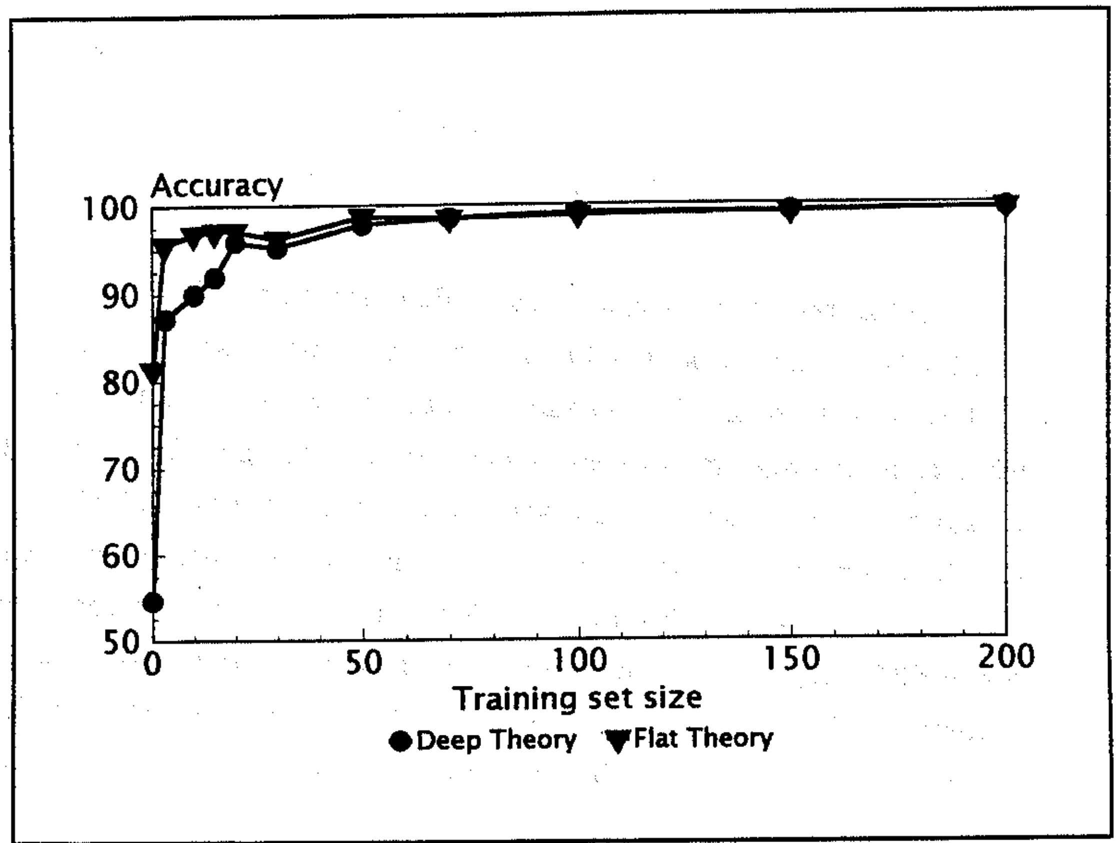

Figure 34. Theory revision on multi-level and single-level theories.

set increases. At a training set size of 200, the average accuracies are equal; there is no longer any statistical difference between the two curves (p > 0.00). Thus, while the structure of the theory affects Forte's revision accuracy for small theories, as the training set size grows this effect diminishes, and it ultimately disappears altogether.

Chapter 7
RESULTS IN LOGIC PROGRAMMING

FORTE represents theories as Prolog programs. Hence, any theory revision is actually an exercise in logic program synthesis or revision. However, few of the standard problems in machine learning require recursion, whereas logic programs written to solve programming problems are almost always recursive. Further, while we may be satisfied with a highly accurate classification theory, we expect programs to be completely correct. Consequently, we view FORTE's operation in this domain differently, and we test it differently. Instead of producing learning curves showing increasing accuracy with larger training sets, we wish to show that, given sufficient training data, FORTE will produce a completely correct program.

Unfortunately, producing an appropriate training set may be more work than writing the correct program in the first place. Hence, applying FORTE to pure program synthesis may not be profitable. However, its ability to perform theory revision can be useful. Consider the case of novices learning to write logic programs. While the instructor may provide model solutions, there are often many ways to solve a problem, and a model solution doesn't help the novices understand how to fix their programs. However, since FORTE tries to correct a theory while preserving as much of the original structure as possible, it can provide novices with customized feedback on what changes need to be made to make their programs work. To test this hypothesis, we presented FORTE with actual programs written by students learning Prolog in a programming languages course. The results of this experiment are discussed in Section 7.2.

All of the programs examined in Sections 7.1 and 7.2 are relatively simple. In particular, FORTE is never asked to revise lower-level recursive predicates. While FORTE can revise such predicates, doing so is inherently difficult, as we discuss in Section 7.3. However, FORTE is able to correct certain types of errors in deeply recursive programs[20], and in Section 7.4 we look at the problem of revising a realistic logic program, namely, a propositional machine-learning algorithm that builds decision trees.

7.1 SYNTHESIS OF LOGIC PROGRAMS

Although designed as a theory revision system, FORTE is able to inductively synthesize simple logic programs from examples of the desired behavior. However, as discussed in Section 5.2.1, in order to correctly synthesize or revise a recursive theory, FORTE requires the training set to provide a complete extensional definition for a subset of the problem domain as well as a representative set of negative instances.

Table I presents a summary of several program synthesis problems that FORTE has been applied to. The first column in the table identifies the program. The second column shows the size of the training set that was provided. The third columns gives the run-time[21] required for the synthesis with relational pathfinding disabled; a dash indicates that FORTE was unable to synthesize a correct, recursive program[22]. The fourth column gives the run-time for the synthesis with relational pathfinding enabled.

[20]By *deeply recursive* we mean theories that contain lower-level recursive predicates.

[21]All run-times in this chapter were generated using a SPARCstation 2.

[22]In these cases, Forte frequently created a series of nonrecursive clauses that are correct on the training data, but do not generalize to unseen instances.

Table I. Summary of program synthesis results.

Program	Training Set Size	Without Relational Pathfinding	With Relational Pathfinding
member	21 instances	4 seconds	4 seconds
append	39 instances	---	21 seconds
directed path	121 instances	25 seconds	24 seconds
insert after	35 instances	30 seconds	50 seconds
merge sort	60 instances	---	199 seconds
naive reverse	38 instances	---	207 seconds

The member program is the standard list utility to determine whether or not the first argument is a member of the list given as the second argument. Similarly, the append program is the standard list utility that appends two lists to produce a third. The directed-path program determines whether or not a path exists in a graph from one node to another. Insert-after inserts an element into a list immediately following a specified marker element, and returns the new list. The merge-sort program is the top-level predicate for a standard merge sort. Finally, the naive reverse program is a two-place reverse program (as opposed to the more efficient three-place reverse that uses an accumulator).

In the sections that follow, we examine the synthesis of the insert-after and merge-sort programs in more detail.

7.1.1 Synthesis of Insert-After

The first example we will examine in detail is the problem of synthesizing a predicate that will insert a new element *n* into a list immediately after the first occurrence of a specified marker element *m*. For this domain, the fundamental domain theory contains only a definition of the list constructor predicate components/3. The training set includes the following positive examples

```
insert_after([m], m, n, [m,n])

insert_after([a,m], m, n, [a,m,n])
insert_after([m,a], m, n, [m,n,a])
insert_after([m,b], m, n, [m,n,b])
insert_after([m,m], m, n, [m,n,m])
insert_after([b,m], m, n, [b,m,n])

insert_after([a,b,m], m, n, [a,b,m,n])
insert_after([a,m,b], m, n, [a,m,n,b])
insert_after([m,a,b], m, n, [m,n,a,b])
insert_after([m,m,m], m, n, [m,n,m,m])
```

In what sense is this a complete extensional definition? Consider the instance

```
insert_after([a,b,m], m, n, [a,b,m,n])
```

When developing a recursive clause in this domain, we may recurse on either the first or the last argument. The only available destructor is list decomposition, i.e., components(List, Head, Tail). Since the predicate contains two lists, we may recurse on either one of them. For recursion on the first argument, we must provide all positive instances of the form

```
insert_after([b,m], m, n, ...)
```

For recursion on the last argument, we must provide all positive instances of the form

```
insert_after(..., m, n, [b,m,n])
```

In this case, both of these requirements are met by the single instance

```
insert_after([b,m], m, n, [b,m,n]).
```

In order to evaluate this instance in turn, we also need to provide

```
insert_after([m], m, n, [m,n])
```

This last instance represents the base case, where the insertion actually occurs. Similar paths to base-cases must exist for all instances in the data set. The simplest way to ensure this is to provide exhaustive examples for problems below a certain size.

In addition, the training set must include a representative set of negatives. The simplest way to generate these is to simply supply all permutations of the positive instances. If this would result in too large of a training set, then the user can instead provide a representative selection of negatives. A representative set of negatives, in this case, includes instances of all sizes for which we have positive instances, and includes all of the various faults that might occur: failing to insert the new element, inserting it in the wrong place, inserting it more than once, and so forth. The exact data set used for these tests appears in the appendices.

When FORTE is presented with these training instances, it first produces the base-case rule

```
insert_after([A|B], A, C, [A,C|B]).
```

The embedded functional list notation is provided by the theory translator for the lists domain, and is strictly for the user's convenience. Internally, FORTE uses the function-free notation

```
insert_after(D, A, C, E) :-
        components(D, A, B),
        components(E, A, F),
        components(F, C, B).
```

Once FORTE has the base-case, it develops the recursive rule

```
insert_after([A|B], C, D, [A|E]) :-
        A \= C,
        insert_after(B, C, D, E).
```

This rule completes the definition of the insert_after predicate. The resulting program is what we might expect a human programmer to write.

7.1.2 Synthesis of Merge_Sort

Not all automatically synthesized programs look like ones a human programmer might produce. Consider the problem of writing a merge_sort predicate, given the standard auxiliary predicates split/3 and merge/3, i.e.,

```
split(List, Split1, Split2) :- List = [_,_|_], split1(List, Split1, Split2).

split1([], [], []).
split1([A], [A], []).
split1([A,B|Rest], [A|Rest1], [B|Rest2]) :- split1(Rest, Rest1, Rest2).

merge(L1, [], L1).
merge([], L2, L2).
merge([H1|L1], [H2|L2], [H1|M]) :- H1 < H2, merge(L1, [H2|L2], M).
merge([H1|L1], [H2|L2], [H2|M]) :- H1 >= H2, merge([H1|L1], L2, M).
```

The training set provides instances for lists of up to length 4 (see the appendices). The base-case FORTE creates is

```
merge_sort(A, A) :- merge(A, B, A).
```

The base cases that would normally be written by a person are

```
merge_sort([], []).
merge_sort([X], [X]).
```

Both of these base-cases are required since split/3 predicate fails if asked to split a list containing fewer than two elements. FORTE's single base case subsumes these two clauses. It states that a list is already sorted if it can be merged with some other list to return itself (of course, the other list is the empty list). Since merge/3 fails if the lists to be merged differ in length by more than one element, this has the effect of requiring list A to be empty or contain only a single element.

FORTE generates the recursive clause using relational pathfinding. Chapter 5 illustrated relational pathfinding on a family domain, where all constants were predefined. For the merge_sort problem, relational pathfinding actually constructs new terms. Suppose relational pathfinding begins with the instance merge_sort([4,3,2,1], [1,2,3,4]). This means that it begins with the two domain constants [1,2,3,4] and [4,3,2,1]. It explores the relational graph by using the relations split/3, merge/3, and merge_sort/2. For example, one path extending from [4,3,2,1] begins with the relation split([4,3,2,1], A, B), where A and B are new variables. The predicate split/3 instantiates these to the values [4,2] and [3,1]. Similarly, one path extending from [1,2,3,4] begins with merge([2,4], [1,3], [1,2,3,4]).

After expanding each node once, however, relational pathfinding is still unable to find an intersection. It requires a second expansion of one of the nodes to develop the relational path shown in Figure 35. While this path successfully links the two original constants, it is still not a valid path. Two of the new constants, [4,2] and [2,4], appear in only one relation. In order to for a relational path to be valid, FORTE must tie up such "loose ends" by adding additional relations. In this case, FORTE completes the path by adding the relation merge_sort([4,2], [2,4]), producing the final clause

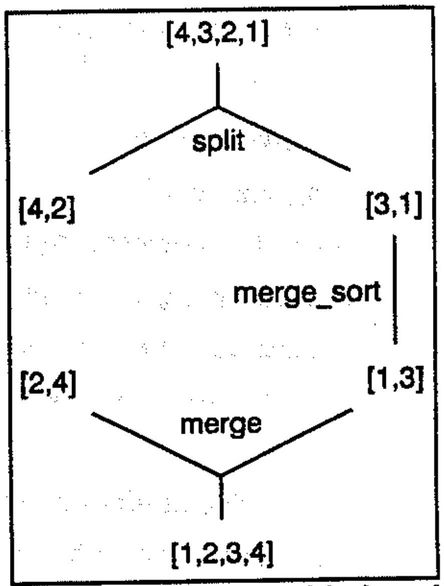

Figure 35. Relational pathfinding of the recursive merge-sort clause.

```
merge_sort(A, B) :-
        merge(C, D, B),
        split(A, E, F),
        merge_sort(E, C),
        merge_sort(F, D).
```

Again, this differs from what a human might write. The order chosen for the antecedents is efficient for FORTE, which works with all ground instances, since it maximizes the number of bound variables at each antecedent. However, a human programmer would know that a sorting predicate is most often called with the second argument unbound, and would therefore place the merge antecedent last rather than first.

7.2 DEBUGGING STUDENT PROGRAMS

Given an incorrect program and a training set containing sufficient positive and negative examples, FORTE should correct the bugs in the program. Working with logic programs provides an opportunity to give FORTE realistic incorrect programs. We asked students in an undergraduate class on programming languages to hand in their first attempts at writing Prolog programs. They gave us their programs after they had satisfied themselves that the programs were correct, but before they tried to run them.

The student programs were distributed among three problems: find a path through a directed graph, insert an element into a list, and merge-sort a list. We collected 23 distinctly different buggy programs, representing a wide variety of errors ranging from simple typographical mistakes to complete misunderstandings of recursion. FORTE was able to debug all of these programs (see Table II and Appendices D-F). The training sets were the same as those used to synthesize these programs in Section 7.1.

Table II. Summary of program revision results.

Program	# of Programs	Train-Set Size	Revision Time	% Correct
directed path	4	121 instances	87 seconds	100%
insert after	9	35 instances	82 seconds	100%
merge sort	10	60 instances	437 seconds	100%

Since FORTE is able to synthesize all of these programs from scratch (see Section 7.1), it is not surprising that FORTE can correct bugs in incorrect versions of these programs. However, the sequence of revisions FORTE makes can be instructive to the novice. Furthermore, since FORTE tries to minimize the changes it makes to a theory, it essentially tries to revise a program along the lines the author intended.

These features make FORTE well-suited for integration into an automated tutoring system for programming languages. The instructor would provide FORTE with suitable training sets for simple programming assignments, and students could ask FORTE to suggest revisions to their programs. It would provide an individualized critique, suggesting only those changes necessary to make their program work correctly—even if it the result differs from the model solution the instructor had in mind.

In the following sections, we discuss a sample revision of each of the student programs. Revisions for all programs appear in the appendices.

7.2.1 Directed Path

To begin, consider the simple case of finding a path through a directed graph. The students were provided with the graph in Figure 36. FORTE was given an exhaustive list of positive and negative instances for this graph. Most of the incorrect student programs for this example were close to the correct program

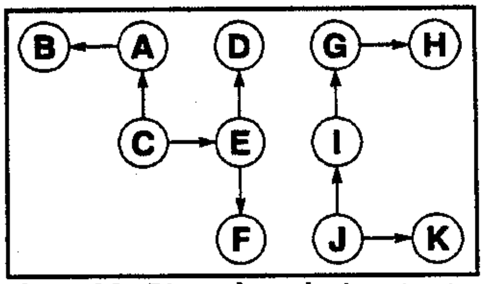

Figure 36. Directed graph given to students and to FORTE.

```
path(A, B) :- edge(A, B).
path(A, B) :- edge(A, C), path(C, B).
```

Hence, FORTE's revisions to them often produce this model program. However, one student's program was

```
path(A, B) :- edge(B, A).
path(A, B) :- edge(A, B).
path(A, B) :- edge(A, C), edge(D, B), path(C, D).
```

In this case, FORTE develops a different revised program. The first clause allows undirected paths; hence it covers a number of negatives and no positives. FORTE retracts this clause as its first revision. Next, the initial program does not work for paths of length 2. Hence, FORTE adds the new rule

```
path(A, B) :- edge(A, C), edge(C, B).
```

to cover these paths. FORTE leaves the recursive rule intact, since it is correct. The final, correct program is

```
path(A, B) :- edge(A, B).
path(A, B) :- edge(A, C), edge(C, B).
path(A, B) :- edge(A, C), edge(D, B), path(C, D).
```

This is not the simple program that FORTE would induce. This revised program has two base cases instead of just one. Both base cases are required since the recursive clause performs a two-step recursion. FORTE preserved the author's original induction scheme, and corrected the erroneous clauses to provide valid base cases.

7.2.2 Merge-Sort

The second type of student program is a merge-sort. The students were asked to write the top-level predicate, given the definitions of split/3 and merge/3 shown in Section 7.1.2. The predicate FORTE synthesizes is

```
merge_sort(A, B) :- merge(A, B, A).
merge_sort(A, B) :-
        merge(C, D, B),
        split(A, E, F),
        merge_sort(E, C),
        merge_sort(F, D).
```

While FORTE's base case is correct, it is hardly intuitive. However, when asked to revise a novice program containing a deficient base case, such as

```
merge_sort([], []).
merge_sort(A, B) :-
        split(A, E, F),
        merge_sort(E, C),
        merge_sort(F, D),
        merge(C, D, B).
```

FORTE produces the simplest revision possible, namely, adding the missing clause

```
merge_sort([A], [A]).
```

to develop a correct program.

7.2.3 Insert-After

The third type of student program was a list-manipulation predicate. Consider the problem of inserting a new element in a list, following a specified marker element. The element may only be inserted once, and only after the first occurrence of the marker (i.e., it is not resatisfiable). The students needed no auxiliary predicates to solve this problem. FORTE, in the fundamental domain theory, used only its standard components/3 list constructor.

The most common error the students made on this program was to forget about the possibility of resatisfaction. A program that will produce a correct result on the first call is

```
insert_after([A|B], A, C, [A,C|B]).
insert_after([A|B], C, D, [A|E]) :- insert_after(B, C, D, E).
```

However, on backtracking, this program can be resatisfied if the marker element occurs more than once in the input list, and the new element will then be inserted in the wrong position. Correcting this error only requires the addition of a single antecedent, to produce the program

```
insert_after([A|B], A, C, [A,C|B]).
insert_after([A|B], C, D, [A|E]) :- A \= C, insert_after(B, C, D, E).
```

One student turned in the program

```
insert_after([A|B], C, D, [A|E]) :- insert_after(B, C, D, E).
insert_after([A|B], A, C, [A,C|D]) :- insert_after(B, A, C, D).
```

This program is both erroneously resatisfiable, and is missing a base case. The first revision FORTE makes is to add the clause

```
insert_after([A|B], A, C, [A,C|B]).
```

Next, the system adds the antecedent A \= C to the first rule, as discussed above. Finally, the last clause is deleted, as it would allow the new element to be inserted more than once in the list.

7.3 REVISION OF DEEPLY RECURSIVE PROGRAMS

When a logic program contains more than one recursive predicate, it is often difficult to assign the blame for failure to one particular predicate. The problem is that an error in a recursive predicate often has a catastrophic effect on theory accuracy. FORTE is able to repair top-level recursive predicates effectively by treating the positive instances in the training set as an extensional definition of the correct predicate, and using this extensional definition to evaluate recursive calls while the predicate is being revised. In order to use the same technique on lower-level recursive predicates, FORTE derives temporary extensional definitions from proofs (or attempted proofs) of the positive instances in the training set, collecting the first call made by each proof to the predicate under revision.

An example using this technique is the following example of the subset predicate, in which we have left the member predicate completely undefined:

```
subset([], A).
subset([A|B], C) :- member(A, C), subset(B, C).

member(A, B).
```

FORTE derives examples for member from the positive instances of subset. Using the resulting set of examples as an extensional definition, FORTE is able to derive the expected definition for member, producing the correct program

```
subset([], A).
subset([A|B], C) :- member(A, C), subset(B, C).

member(A, [A|B]).
member(A, [B|C]) :- member(A, C).
```

The approach of deriving extensional definitions is not foolproof, but is often effective. The method fails in three primary circumstances. The first occurs when the top-level predicate is itself so incorrect that it does not provide a meaningful set of calls to the lower-level predicate. For example, proofs of positives might usually fail before the target predicate is even called. In this case, the extensional definition for the lower-level predicate will be too small and incomplete to be of any use. Or the call to the lower-level predicate may be incorrect in a way that prevents FORTE from developing any reasonable definition for it, even an unintended one. An example of this problem occurs with the following incorrect program for subset:

```
subset([], A).
subset([A|B], C) :- member(A, B), subset(B, C).

member(A, B).
```

In this case, member is being called with the head and tail of the same list. Since it doesn't even have access to the second list, any predicate FORTE develops for member can only reflect spurious correlations in the structure of the first list. In this example, FORTE actually decides not to use member at all, but to develop a correct but incomplete definition for subset alone:

```
subset([], A).
subset(A, [B|A]).
subset([A|B], [A|C]) :- subset(B, [A|C]).
```

The second common case in which our method of deriving extensional definitions fails is when the calls to the lower level predicate are not ground. In this case, the derived extensional definition is overly general, and FORTE is likely to develop an unintended definition for the lower-level predicate. This occurs when revising the following program, where sublist/2 is a predicate designed to succeed when the first argument is a sublist of the second argument.

```
sublist(A, B) :- append(A, C, B).
sublist(A, [B|C]) :- sublist(A, C).

append([], A, A).
append([A|B], C, [A|D]).
```

The top-level call to append contains an uninstantiated argument, leading to extensional facts of the form append([a,b], X, [a,b,c]). Nonetheless, FORTE is able to revise the definition of append to produce a working sublist predicate:

```
append([], A, A).
append([A|B], C, [A|D]) :- append(B, D, D).
```

This predicate has the required capabilities for its role in the sublist predicate, namely, it succeeds whenever the first argument is a prefix of the third. However, it should not be called append. Since the second argument was unconstrained, it took on a meaning different from what one expects in an append predicate.

Another common cause of uninstantiated calls to a lower-level predicate is a procedural program where one predicate produces a value which is later consumed by another predicate. For example, consider the merge_sort predicate

```
merge_sort(A, B) :-
        split(A, C, D),
        merge_sort(C, E),
        merge_sort(D, F),
        merge(E, F, B).
```

The predicate split/3 "produces" lists C and D for consumption by the recursive calls to merge_sort/2. Since this means that split/3 is called with these variables uninstantiated, it would be difficult for FORTE to correctly revise it.

The third common failure occurs when the lower-level predicate we wish to revise is called with a restricted set of arguments. In this case, the program we learn may actually be correct, but the lower level predicate does not have the expected meaning. This can occur, for example, when revising the naive reverse program, since reverse always calls append with lists of length one in the second argument. Suppose we begin with the incorrect program

```
reverse([], []).
reverse([A|B], C) :- append(D, [A], C), reverse(B, D).

append([A|B], C, [D|E]) :- append(B, C, E).
```

FORTE successfully revises this program to be a correct implementation of reverse:

```
reverse([], []).
reverse([A|B], C) :- append(D, [A], C), reverse(B, D).

append(A, [B|A], [B|A]).
append([A|B], C, [A|D]) :- append(B, C, D).
```

The definition of append/3 is correct for its role in this program. However, it is not a general-purpose append predicate. The first clause must require A to be the empty list. The definition given works, but only if the second argument is a list of only one element. This is always true for this program, but is not true in general.

7.4 DECISION-TREE INDUCTION

In order to demonstrate FORTE's potential as an automated debugging system for logic programs, we presented it with buggy versions of a large, realistic logic program. This program is a variation[23] of the decision-tree induction program in [Bratko, 1991]. The previous section describes the limitations of FORTE's ability to revise deeply-recursive programs, and there are many predicates in this program that FORTE cannot revise, primarily because they are procedural in nature (the producer-consumer problem where predicates are called in non-ground mode). Additionally, certain errors in higher-level predicates cause FORTE to be unable to develop adequate training sets for lower-level recursive predicates. However, FORTE is able to repair many realistic bugs in this program.

7.4.1 Methodology

We made one concession to efficiency when introducing bugs, in that we placed most of the program's lower-level predicates into the fundamental domain theory. Since the fundamental domain theory is compiled rather than meta-interpreted, and since FORTE does not explore revision points in the shielded predicates, this substantially increased the speed of revisions. This also allows us to improve efficiency by using Prolog built-in functions such as nonvar and cut in some of the lower-level predicates. The fundamental domain theory and revision verifier were otherwise empty. The language bias was set to allow recursion, and antecedents from the theory, fundamental

[23]The primary difference between this program and the original is that two mutually recursive predicates have been collapsed into the single recursive predicate induce_tree/5, since FORTE cannot revise programs containing mutual recursion.

domain theory, and built-in equality relations. The portion of the program FORTE was asked to revise is shown below. The complete program appears in Appendix G.

```
/* Top-level: given attributes and examples, produce decision tree */
induce_tree(Attr, Examples, Tree) :-
        induce_tree(A1, [], Attr, Examples, Tree).

/* Given no examples, produce an empty tree */
induce_tree(_, [], _, [], [] ).

/* If examples are pure, the tree is the class of the examples */
induce_tree( A1, [], A2, Examples, Class ) :-
        Examples = [Example|_],
        pure(Examples),
        Example = [Class, _].

/* If the examples are impure, choose best attribute as root of subtree */
induce_tree( _, [], Attributes, Examples, [Attr_name, Subtrees] ) :-
        Examples = [_|_],
        impure(Examples),
        choose_attribute( Attributes, Examples, 0, _, Attribute),
        fdt_delete( Attribute, Attributes, Rest_atts),
        Attribute = [ Attr_name, Values ],
        induce_tree( Attr_name, Values, Rest_atts, Examples, Subtrees).

/* create subtree branch for each value in selected attribute */
induce_tree(Name, [Val|Vals], Rest_atts, Examples, [[Val, Tree] | Trees]) :-
        Vals = [_|_],
        attval_subset(Name, Val, Examples, Example_subset),
        induce_tree(A1, [], Rest_atts, Example_subset, Tree),
        induce_tree(Name, Vals, Rest_atts, Examples, Trees).

/* last subtree branch */
induce_tree(Name, [Val], Rest_atts, Examples, [[Val, Tree]]) :-
        attval_subset(Name, Val, Examples, Example_subset),
        induce_tree(A1, [], Rest_atts, Example_subset, Tree).
```

```
/* choose attribute yielding highest purity */
choose_attribute( [], _, _, Best, Best).
choose_attribute( [Attr|Attrs], Examples, Best_purity, Best_so_far, Best) :-
        purity(Attr, Examples, Purity),
        less(Best_purity, Purity),
        choose_attribute( Attrs, Examples, Purity, Attr, Best ).
choose_attribute( [Attr|Attrs], Examples, Best_purity, Best_so_far, Best) :-
        purity(Attr, Examples, Purity),
        less_or_equal(Purity, Best_purity),
        choose_attribute( Attrs, Examples, Best_purity, Best_so_far, Best ).
```

There were 26 training instances, 14 positive and 12 negative. A single training instance for FORTE included the attributes and instances given to the decision-tree program along with the decision tree expected as output. All FORTE instances were based on the set of attributes and decision-tree instances given below; the correct decision tree associated with these instances is shown in Figure 37. Positive instances given to FORTE included the whole decision tree, all subtrees thereof, and repetitions of some instances with the attributes listed in a different order.

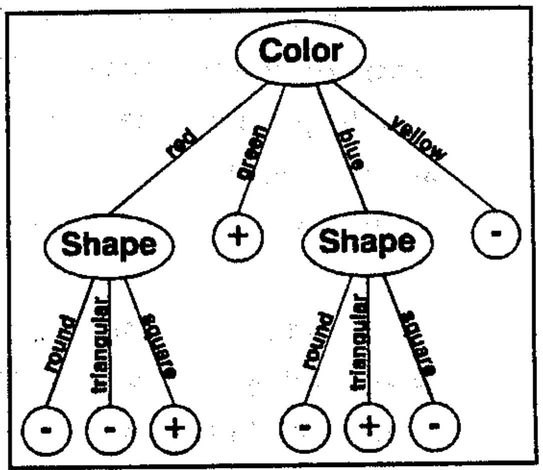

Figure 37. Decision tree induced by the target program.

<u>Attributes</u>
shape: round, triangular, square
color: red, green, blue, yellow
size: small, large

Positive Instances	Negative Instances
triangular, green, large	round, red, small
square, green, large	square, blue, small
square, red, large	round, yellow, large
round, green, small	triangular, red, small
triangular, blue, large	round, blue, large
	triangular, yellow, small
	square, yellow, small

7.4.2 Sample Revisions

In this subsection we show two sample corruptions of the theory that FORTE successfully repaired. Both of these corruptions introduced two errors into the program, one at the second level recursive predicate induce_tree/5 and another at the third-level recursive predicate choose_attribute/5. These are the two predicates that drive the decision-tree algorithm.

In the first example, a new rule is added to induce_tree/5. This rule is an overly-general version of the second clause in the correct program. In choose_attribute/5, an antecedent from the second clause is deleted. The incorrect clauses are:

```
induce_tree( A1, [], A2, Examples, Class ) :-                    /* New Rule */
        Examples = [Example|_],
        Example = [Class, _].

choose_attribute( [Attr|Attrs], Examples, Best_purity, Best_so_far, Best) :-
        purity(Attr, Examples, Purity),
        /* less(Best_purity, Purity),                 Antecedent Deleted */
        choose_attribute( Attrs, Examples, Purity, Attr, Best ).
```

FORTE repaired these errors, exactly restoring the original theory. The revision took 78 minutes on a SPARCstation 2. This relatively long revision time is due to two factors. First, the length of the initial theory leads to many possible revision points. Second, the high arity of the predicates and large number of variables in each clause leads to a large number of possible revisions at each revision point.

The second example corrupted the same two key predicates, but in a different way. The second rule in induce_tree/5 contains an incorrect antecedent (the correct one was deleted and a new one added in its place). And the base-case of choose_attribute/5 is overly general, due to the addition of an anonymous variable. The incorrect predicates were:

```
induce_tree( A1, [], A2, Examples, Class ) :-
        Examples = [Example|_],
        impure(Examples),                        /* Incorrect Antecedent */
        Example = [Class, _].

choose_attribute( _, _, _, Best, Best).          /* Overly General Base Case */
```

FORTE repaired these errors, restoring the original correct program in 50 minutes on a SPARCstation 2.

7.5 COMPARISON TO PDS6

One of the most well-known Prolog debugging systems is Shapiro's PDS6 [Shapiro, 1983]. We took an annotated PDS6 test from [Murray, 1986] and gave the same problem to FORTE. The incorrect program is a quicksort containing three errors:

```
qsort([X|L], L0) :-
        partition(L, X, L1, L2),
        qsort(L1, L3), qsort(L2, L4),
        append(L3, [X|L4], L0).

partition([X|L], Y, L1, [X|L2]) :-
        partition(L, Y, L1, L2), less(X, Y).
partition([X|L], Y, [X|L1], L2) :-
        partition(L, Y, L1, L2), less_or_equal(X, Y).
partition([], X, [], []).

append([X|L1], L2, L3) :-
        append([X|L1], L2, [X|L3]).
append([], L, L).
```

One error appears in each of the predicates. The base-case for qsort/2 is missing. The variables are reversed in less/2, in the first partition/4 clause. Finally, the recursive append/3 clause is wrong. The corrected program developed by PDS6 is:

```
qsort([X|L], L0) :-
        partition(L, X, L1, L2),
        qsort(L1, L3), qsort(L2, L4),
        append(L3, [X|L4], L0).
qsort([], []).

partition([X|L], Y, L1, [X|L2]) :-
        partition(L, Y, L1, L2), less(Y, X).
partition([X|L], Y, [X|L1], L2) :-
        partition(L, Y, L1, L2), less_or_equal(X, Y).
partition([], X, [], []).

append([X|L1], L2, [X|L3]) :-
        append(L1, L2, L3).
append([], L, L).
```

FORTE is unable to correctly revise the erroneous program. However, when the errors appear singly, FORTE is able to correct all but the erroneous append clause. The reason that this clause is not correctly revised is that FORTE evaluates recursive calls last. This means that append is called with two unbound variables, which leads to an inadequate derived training set.

Chapter 8
RESULTS IN QUALITATIVE MODELLING

To demonstrate FORTE's ability to work in diverse domains, we have applied it in the constraint-based domain of qualitative modelling. When supplied with appropriate domain knowledge through the fundamental domain theory and the revision verifier, FORTE is able to synthesize and revise qualitative models suitable for use by QSIM [Kuipers, 1986].

In order to provide some background for the results in the chapter, Section 8.1 provides a brief description of the qualitative modelling task as implemented by QSIM. Section 8.2 describes the representation FORTE uses in this domain and provides a summary of results in this domain. The remaining sections present a detailed look at particular qualitative modelling problems.

8.1 QUALITATIVE SIMULATION

QSIM [Kuipers, 1986] is a qualitative simulation system that takes as input a qualitative model of a system along with an initial state and produces as output a complete set of system behaviors. The premise of qualitative simulation is that it is possible to describe the general behavior of a system without knowing anything about the numerical values involved. For example, given a bathtub with water flowing into it and draining out of it, there are three possible behaviors: it can stay empty (very large drain), it can overflow, or it can reach an equilibrium where the water level rises to a point where the water flows out as fast as it flows in.

QSIM uses qualitative differential equations (QDEs) to describe systems. Using FORTE's syntax, the QDE for the bathtub would be

```
bathtub(Amount, Inflow, Outflow, Netflow) :-
        constant(Inflow),
        derivative(Amount, Netflow),
        add(Netflow, Outflow, Inflow),
        m_plus(Outflow, Amount).
```

The first constraint states that Inflow is constant—it does not change over time. The second constraint, derivative(Amount, Netflow), indicates that Netflow is the derivative of amount. This means, for example, that, if Netflow is positive, then Amount must be increasing. The third constraint says that the sum of Netflow and Outflow yields Inflow (or, more intuitively, Netflow is the difference of Inflow and Outflow). Finally, the last constraint indicates that some monotonically increasing function holds between Outflow and Amount—i.e., the more water there is in the bathtub, the faster it flows out.

Given this QDE, and initial values for the variables, QSIM produces behaviors for the system. A behavior contains, for each system variable, a time-ordered sequence of values, directions of change, and signs. Values are symbols whose only meaning is in their relative ordering; there are three predefined values: minus-infinity, 0, and infinity. Directions of change may be decreasing (↓), steady (⊖), or increasing (↑). Signs are minus (-), zero (0), and plus (+). As an example, the equilibrium behavior for the bathtub is shown in Table III.

In this behavior, Amount begins with a value of 0, it is increasing, and its sign is 0. At time 2, Amount has a value in the open interval between 0 and max (the predefined maximum volume of the bathtub), it is increasing, and its sign

Table III. Behavior showing a simple bathtub reaching equilibrium

Variable	Time 1	Time 2	Time 3
Amount	0, ↑, 0	(0, max), ↑, +	equil_amt, ⊖, +
Inflow	in_1, ⊖, +	in_1, ⊖, +	in_1, ⊖, +
Netflow	net_1, ↓, +	(0, net_1), ↓, +	0, ⊖, 0
Outflow	0, ↑, 0	(0, inf), ↑, +	equil_out, ⊖, +

is positive. At time 3, Amount has a value of equil_amt, it is steady, and its sign is positive. We know that equil_amt is less than max, since Amount began at 0 and did not pass through max. Similar explanations apply to the other variables shown.

8.2 FORTE AND QUALITATIVE SIMULATION

FORTE's task is to induce or revise qualitative models, given instances of behaviors. A qualitative model is a single clause, which means that the language bias will be conjunctive. Furthermore, one generally wants tightly constrained models that produce only the desired behaviors, so the language bias will also be most-specific. This means that FORTE will produce a single clause containing all constraints consistent with the input behaviors. As discussed in [Richards, Kraan, and Kuipers, 1992], when given complete behavioral information, a model generated in this manner is guaranteed to be unique, complete, and correct.[24]

We provide system behaviors to FORTE as complex functional terms. For example, the behavior of Amount for the bathtub above is given as

[24]The definition of complete behavioral information in [Richards, Kraan, and Kuipers, 1992] has recently been found to be incomplete. However, this deficiency does not affect the results in this thesis.

```
[amount,     [0, max, inf],
             [[mass],[]],
             [[0, inc, 0],[[0,max], inc, plus],[equil_amt, std, plus]]]
```

The first term gives the space of qualitative values; in this case, amount may range from 0 through max to infinity. The second term provides the dimensions of amount. The first sublist is the numerator and the second is the denominator. In a quantitative simulation dimensions would be some unit such as kilograms, but in a qualitative domain it is the more abstract unit *mass*. The third term provides the actual behavioral information from Table III.

Of course, theories in FORTE do not include functions, so FORTE is unaware of the internal structure of these terms. Instead, the fundamental domain theory provides definitions of the QSIM constraints, and these constraints are responsible for interpreting the behavioral information. For example, when FORTE tries to prove the constraint m_plus(Amount, Outflow), for the above behavior, the variables Amount and Outflow are instantiated to the appropriate behavior terms and FORTE calls the fundamental domain theory predicate m_plus/2. Predicate m_plus/2 must then determine whether or not a monotonically increasing function holds between the two given behaviors. A more complete description of QSIM constraints is beyond the scope of this chapter, but may be found in [Kuipers, 1986, 1989]. FORTE's implementation of the QSIM constraints is taken from MISQ ([Kraan, Richards, and Kuipers, 1991] [Richards, Kraan, and Kuipers, 1992]), and includes: constant, M+, M-, add, mult, and derivative. From FORTE's point of view, however, the constraints in the fundamental domain theory are simply antecedents that succeed or fail in the course of a proof.

Also, the constraints can instantiate variables. In the family domain, calling parent(fred, X) would instantiate X to a child of fred. Here, calling m_plus(Amount, X) will instantiate X. However, since this is a constraint-based domain, X will only be partially instantiated, to constrain it to the set of possible behaviors that are monotonic functions of Amount. This means, for example, that Amount and X must have the same directions of change at all times, but it says nothing about the values or dimensions of X. The importance of this becomes apparent when we consider how relational pathfinding works in this domain.

Relational pathfinding explores the graph of domain values surrounding the values that appear in a positive instance. In qualitative modelling, nodes in this graph are behaviors and edges are constraints. Thus, to explore the graph, relational pathfinding tries a known behavior in all possible constraints to see what behaviors it can derive. Since the derived behaviors are only partially instantiated, each of them actually stands for a set of possible behaviors. An intersection is reached when two such behavior sets intersect.

To illustrate this, consider the problem of modelling a ball thrown into the air. Suppose that the user only provides behavioral information for position and gravity. The behavior provided appears in Table IV.

Table IV. Behavior of a thrown ball, omitting velocity. Abbreviations: f = floor, h = hand, m = max, c = ceiling. The notation (hc) indicates the open interval from h to c.

Variable	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7
Position	h ↑ +	(hc) ↑ +	m ⊖ +	(hm) ↓ +	h ↓ +	(0h) ↓ +	f ↓ +
Gravity	g ⊖ -	g ⊖ -	g ⊖ -	g ⊖ -	g ⊖ -	g ⊖ -	g ⊖ -

This behavior shows a ball being thrown from someone's hand, rising to a maximum before hitting the ceiling, and then falling to the floor. Gravity is shown as a constant throughout the behavior. FORTE can find no constraints that hold between the two variables. However, when it proposes an empty model, the revision verifier rejects it, since the variables in the model are not connected (the functions of the revision verifier are described fully below), and FORTE attempts to join the variables using relational pathfinding.

One path leading from Position is derivative(Position, V_p)., and one path leading from Gravity is derivative(V_g, Gravity). The derivative constraint defines a relationship between the sign of one variable and the direction of change of the other. The resulting partial instantiations appear as the first two lines in Table V. These two partially instantiated terms can be unified (i.e., the intersection of the behavior sets that they represent is not empty), and their units are consistent[25] so relational pathfinding creates a path containing the two derivative constraints, and intersecting at the new variable whose behavior is shown as the third line of Table V.

[25]The dimensions of Position and Gravity are distance and distance/time^2 respectively, so both V_p and V_g have dimensions of distance/time.

Table V. Two partially instantiated behaviors, and the derived variable resulting from their unification.

Variable	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7
V_p	? ? +	? ? +	0 ? 0	? ? -	? ? -	? ? -	? ? -
V_g	? ↓ ?	? ↓ ?	? ↓ ?	? ↓ ?	? ↓ ?	? ↓ ?	? ↓ ?
Velocity	? ↓ +	? ↓ +	0 ↓ 0	? ↓ -	? ↓ -	? ↓ -	? ↓ -

Thus, FORTE produces the final, correct model

```
ball(Position, Gravity) :-
        derivative(Position, Velocity),
        derivative(Velocity, Gravity).
```

As mentioned above, the revision verifier also plays a role in this domain. The revision verifier allows the user to provide domain-dependent consistency checks. In particular, it rejects a model if not all system variables are used in the constraints or if the model does not form a connected graph. However, if relational pathfinding is unable to connect the model within the depth bound given in the language bias, FORTE will give a warning message and provide the best model it was able to generate.

It should be noted that providing a model that is not connected is not necessarily an error. FORTE brings a new capability to qualitative model building—namely, the ability to create new system variables. However, there is an essential tension between the desire to relate two behaviors and the need to recognize when truly independent process are taking place. On one extreme, any two behaviors can be related by introducing a large enough set of intermediate variables. On the other extreme, two similar behaviors may, in fact, be unrelated. The depth-bound the user provides sets a point in this spectrum—it identifies the maximum number of intermediate variables FORTE is allowed to introduce in an attempt to join to apparently disparate behaviors. If FORTE presents a disconnected model, this is a statement that the processes are independent, within this bound.

Table VI. Summary of qualitative model induction results.

Model	Training Set Size	Number of Constraints	Execution Time
thrown ball (missing velocity)	1 behavior	2 constraints	4 seconds
bathtub	1 behavior	3 constraints	2 seconds
two tubs	2 behaviors	6 constraints	35 seconds
cascaded tanks	1 behavior	9 constraints	10 seconds
cascaded tanks (missing netflows)	2 behaviors	7 constraints	43 seconds
shuttle reaction control system	1 behavior	55 constraints	114 seconds

Table VI provides a summary of the tests run in qualitative modelling. The thrown ball has already been discussed. The bathtub is a simple tank with water flowing in from some source and out through a drain. The two-tubs problem presented FORTE with two independent bathtubs and asked it to build a single model; FORTE correctly refused to connect the two tubs, and provided a model containing two copies of the simple bathtub. The cascaded tanks examples are discussed in detail in Section 8.3. The shuttle reaction control system is discussed in Section 8.4.

8.3 TWO CASCADED TANKS

Cascading two tanks so that the drain from one provides the inflow to the next provides a moderately complex second order system. A correct model for this system appears in Figure 39. FORTE is able to induce this model from a single qualitative behavior. FORTE's induced model contains two redundant constraints, but is otherwise identical to the expected model. The variables have been renamed for easy readability:

```
model(In_A, Net_A, Out_A, Amt_A, Net_B, Out_B, Amt_B) :-
        constant(In_A),
        add(Out_A, Net_A, In_A),
        add(Out_B,  Net_B, Out_A),
        derivative(Amt_B, Net_B),
        derivative(Amt_A, Net_A),
        m_plus(Amt_A, Out_A),
        m_minus(Amt_A, Net_A),
        m_minus(Out_A, Net_A),
        m_plus(Amt_B, Out_B).
```

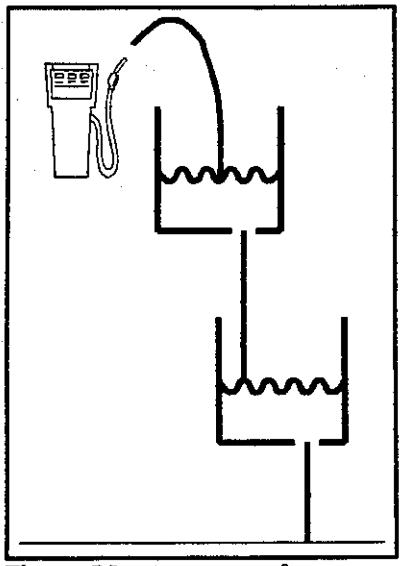
Figure 38. A system of two cascaded tanks.

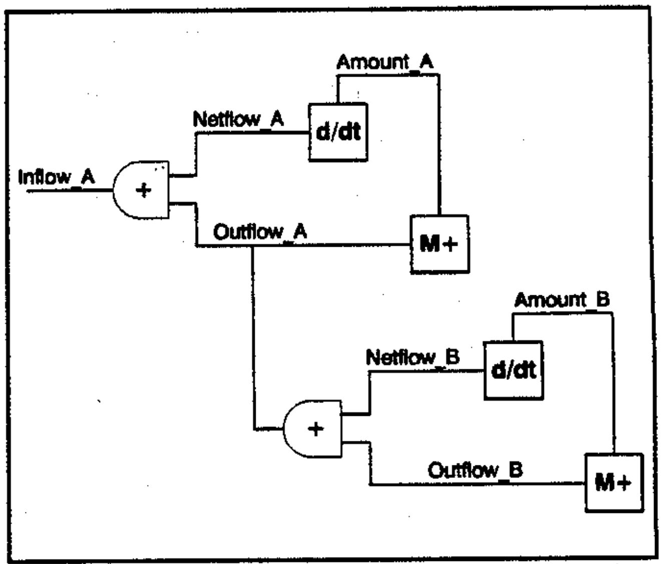

Figure 39. Graphical representation of a correct qualitative model for two cascaded tanks.

In order to test FORTE's ability to induce missing system variables, we omitted those that a user might realistically forget. We supposed the user measured all the flows and amounts but did not realize that the calculated netflow for each tank would be important. Thus, we provided behaviors for all of the other variables, but omitted the netflows entirely. For this test we provided two qualitative behaviors, and inflow_a was not held constant.

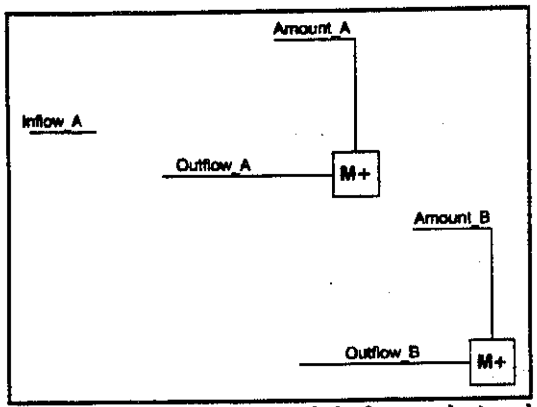

Figure 40. Partial model before relational pathfinding.

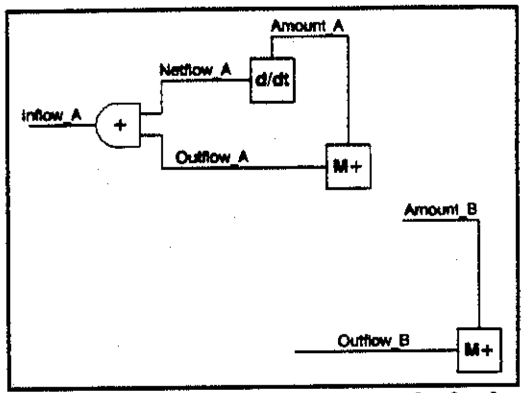

Figure 41. Model after one round of relational pathfinding.

The process FORTE goes through to create a system model in this case can be viewed in three stages. First, before doing any relational pathfinding, it generates the disconnected model shown in Figure 40. The revision verifier rejects this model. After one round of relational pathfinding, FORTE has hypothesized one netflow variable, with the results shown in Figure 41. This model is still not connected, so the revision verifier rejects it again. Finally, after a second round of relational pathfinding, FORTE generates the complete model shown above in Figure 39.

8.4 SHUTTLE REACTION CONTROL SYSTEM

The space shuttle reaction control system (RCS) (see [Kay, 1992]) is substantially more complex than the system of cascaded tanks, and provides a more realistic test of FORTE's capabilities in this domain. The basic functional element of the reaction control system is shown in Figure 42. The system works as follows. Helium from the helium tank enters the fuel tank at a constant, regulated pressure (helium in the fuel tank is referred to as *ullage*). This forces fuel out of the fuel tank and into the manifold. From the manifold, the fuel enters the thruster and ignites to provide thrust.

The complete system, of course, is more complex than this. There are three independent reaction control systems on the shuttle, each of which has both fuel and oxidizer subsystems. The diagram in Figure 42 may be viewed as one fuel or oxidizer subsystem. Within a subsystem, there are redundant valves, and the upper valve is actually a pressure regulator. In addition, the single manifold shown is an abstraction of five parallel manifold chambers leading to five thrusters. However, one can use this element of the RCS to understand the basic properties of the entire system. For the purposes of this section, we will assume that the lowest valve, leading to the thruster, is closed (i.e., the thruster is off). Figure 43 shows the hand-generated model for this system, taken from [Kay, 1992].

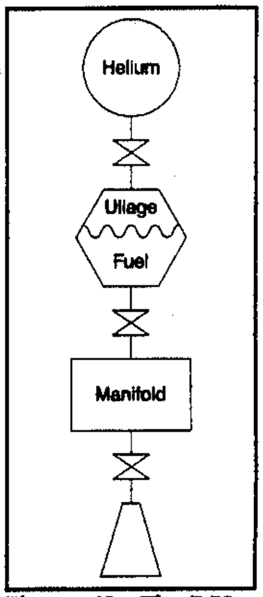

Figure 42. The RCS.

8.4.1 Model Induction

Assume that the upper valve, between the Helium tank and the fuel tank is open and providing constant pressure, and that the center valve between the fuel tank and the manifold has just been opened. If the initial pressure in the manifold is lower than the initial pressure in the fuel tank (so that the system is not immediately at equilibrium), then the fuel flows from the fuel tank into the manifold. Providing this single behavior to FORTE allows FORTE to induce exactly the model shown in Figure 43, with the addition of several redundant constraints.[26]

[26]For example, since Vol_total is constant, one redundant constraint is an M- between Vol_Ull and Vol_Fuel.

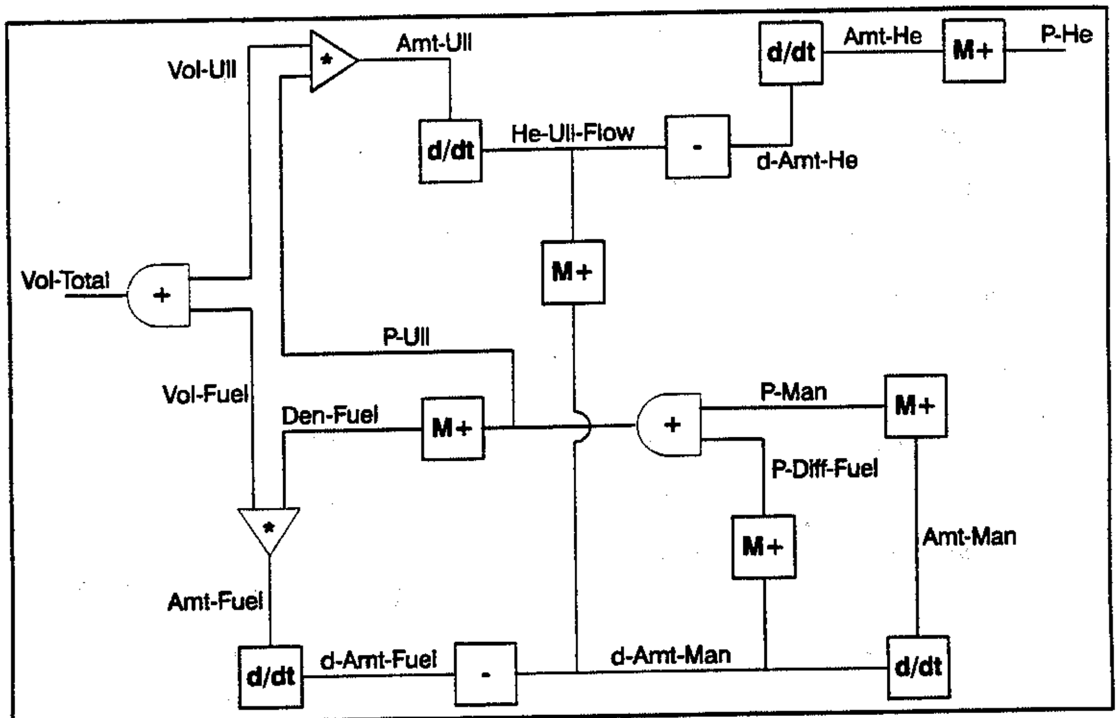

Figure 43. Qualitative model of the RCS.

8.4.2 Model Revision and Diagnosis

Alternatively, suppose the user has a faulty model of the RCS, along with correct system behaviors. Since FORTE is a theory revision system, it can take the faulty model as an initial theory, and revise it to match the correct system behaviors. For example, suppose the user mistakenly used a multiply constraint in place of the add constraint leading to Vol_total in the correct model. FORTE begins by deleting this constraint (or any constraint which keeps the positive instances from being provable). Then, since the model is required to be most-specific, FORTE adds all constraints consistent with the behaviors which do not already appear in the model. The result is, of course, the same model that Forte would induce in the absence of the initial theory.

However, one can view this process of model revision in a different light. Suppose that the user has a correct system model, but that the system itself is behaving incorrectly. In this case, we can use theory revision to revise the correct system model to reflect the actual system behavior. The resulting changes in the model can be viewed as a diagnosis.

One of the failures that can occur in the RCS is a leak in one of the five manifolds leading from the fuel tank. In order to isolate the leak, the astronauts shut the valve leading from the fuel tank into the manifolds. They then isolate the suspected manifold and reopen the valve connecting the fuel tank and the manifolds. If the leak has been eliminated, the system will quickly reach equilibrium. If the leak has not been isolated, the system will not reach a pressure equilibrium (at least, not before all of the fuel has drained out through the leak).

If FORTE begins with the correct system model shown in Figure 43, along with the system behavior caused by a leak in the manifold, FORTE revises the model by deleting the constraint minus(D_Amt_Fuel, D_Amt_Man). The variable D_Amt_Fuel is the amount of fuel leaving the fuel tank and flowing into the manifold. Variable D_Amt_Man is the net change in the amount of fuel in the manifold. Normally, the amount of fuel flowing out of the fuel tank (D_Amt_Fuel) should be the same, except for sign, as the net amount of fuel being added to the manifold. Since FORTE deletes this constraint, there must be another influence on the amount of fuel in the manifold—namely, a leak.

Unfortunately, there are two classes of faults in the RCS that FORTE cannot diagnose. First, operation of the valves controls transitions[27] between different system models, and failures of the valves cause failures in those

[27]A *transition* is a discontinuity in system behavior that requires a new and different model to continue simulation. For example, when modelling the temperature of water, transitions will occur when the water freezes or boils. In the RCS, transitions occur when valves open or close, when a tank becomes completely empty, or when pressure regulation of the Helium fails.

transitions. While FORTE could maintain different system models as different clauses in a theory, the support code from MISQ that constitutes the fundamental domain theory and revision verifier is not designed to handle model transitions.

The second class of faults that FORTE cannot diagnose in the RCS are leaks that occur while the thruster is activated. The reason for this is that the thruster itself acts exactly like a leak. Additional leaks cause no qualitative change in system behavior. Thus, in order to detect leaks while the thruster is operating, we must have access to quantitative information.

Chapter 9
RESULTS IN GRAMMAR ACQUISITION

In this chapter we take a brief look at FORTE's potential in the field of grammar acquisition. FORTE represents theories as pure Prolog programs, and pure Prolog programs can, by a simple syntactic transformation, be turned into definite clause grammars[28] (DCGs). Consequently, since FORTE can revise logic programs, it can also revise DCGs. We illustrate this capability with several revisions of one example grammar.

9.1 METHODOLOGY

The data set for grammar revision includes both positive and negative instances of correct sentences, along with a dictionary of words that provides the part-of-speech and, for nouns and verbs, whether the word is singular or plural. The tests presented in this chapter use the following words:

big:	adjective	feral:	adjective
little:	adjective	ugly:	adjective
a:	determiner, singular	the:	determiner, singular/plural
boy:	noun, singular	boys:	noun, plural
cat:	noun, singular	cats:	noun, plural
dog:	noun, singular	dogs:	noun, plural
chases:	verb, singular	chase:	verb, plural
hits:	verb, singular	hit:	verb, plural

[28]A definite clause grammar is less powerful than a full context-sensitive grammar, but more powerful than a context-free grammar, since nonterminals may take arguments.

In keeping with DCGs, dictionary entries are coded as difference lists, e.g.,

```
adjective( [big|Rest], Rest )
noun( [boy|Rest], Rest, singular )
```

The revision verifier is empty. The fundamental domain theory provides no new predicates. However, it checks the calls to the dictionary to ensure that one of the first two arguments is instantiated—calls with neither of the first two arguments instantiated are useless since they do not connect to other parts of the sentence through the difference lists.

The training set for the revisions shown below includes four correct sentences as positive instances and four incorrect sentences as negative instances. The correct sentences are:

The little boys hit the big dog.
The little boy hits the big dog.
The dog chases the cat.
The big, ugly dogs chase the feral cat.

The incorrect sentences are:

The little boys hits the big dog.
A little boys hit the big dog.
The little boy hits the the dog.
The little the boy hits the dog.

9.2 SAMPLE REVISIONS

Consider the following simple English grammar, which we show using standard Prolog syntax. This grammar structures sentences as noun-phrases followed by verb-phrases, it allows articles and adjectives, and it enforces agreement on number between the subject and the verb.

```
sentence(S, R) :- noun_phrase(S, S1, N), verb_phrase(S1, R, N).

noun_phrase(I, O, N) :- determiner(I, A, N), adj_noun(A, O, N).
noun_phrase(I, O, N) :- adj_noun(I, O, N).

adj_noun(I, O, N) :- adjective(I, A), adj_noun(A, O, N).
adj_noun(I, O, N) :- noun(I, O, N).

verb_phrase(I, O, N) :- verb(I, O, N).
verb_phrase(I, O, N) :- verb(I, A, N), noun_phrase(A, O, M).
```

Revision 1. If the top-level rule for sentence is deleted, leaving only the trivial

```
sentence(A, B).
```

FORTE successfully reconstructs the rule

```
sentence(A, B) :- noun_phrase(A, C, D), verb_phrase(C, B, D).
```

Revision 2. If the lower-level predicate verb_phrase/3 is changed to

```
verb_phrase(I, O, N) :- verb(I, O, N).
verb_phrase(I, O, N) :- verb(I, A, N).
```

FORTE successfully revises it to the original

```
verb_phrase(I, O, N) :- verb(I, O, N).
verb_phrase(I, O, N) :- verb(I, A, N), noun_phrase(A, O, M).
```

Revision 3. If the lower-level predicate verb_phrase/3 is changed to

```
verb_phrase(I, O, N) :- verb(I, O, N).
```

FORTE successfully revises it to

```
verb_phrase(I, O, N) :- verb(I, O, N).
verb_phrase(I, O, N) :- verb(I, A, N), noun_phrase(A, O, M).
```

Revision 4. If the lower-level recursive predicate adj_noun/3 is changed to

```
adj_noun(I, O, N) :- adjective(I, A), adj_noun(A, O, N).
```

FORTE successfully revises it to

```
adj_noun(I, O, N) :- adjective(I, A), adj_noun(A, O, N).
adj_noun(I, O, N) :- noun(I, O, N).
```

However, FORTE is unable to repair errors in the recursive clause of this predicate, since this predicate is called in non-ground mode (see discussion of deeply recursive predicates in Chapter 6).

Chapter 10
RELATED WORK

This chapter compares FORTE to related work in the field. The organization of this section parallels that of Chapter 2. However, where Chapter 2 provided a broad view of related work to set the stage for FORTE, this chapter focuses on specific work that is closely related to FORTE, and to which we wish to contrast FORTE's approach. Related work in machine learning is discussed in the sections on propositional and first-order learning. The following sections discuss related work in the target domains of logic programming and qualitative modelling.

10.1 Propositional Theory Revision

The most closely related propositional theory revision system is EITHER, described in [Ourston and Mooney, 1990]. EITHER is a propositional theory revision system; however, FORTE draws many of its basic theory revision operators from EITHER. EITHER's approach to theory revision takes advantage of the relative tractability of propositional logic and uses a global greedy-search algorithm that guarantees convergence on the training set. FORTE, on the other hand, uses a hill-climbing method that is not guaranteed to converge on the training set. This hill-climbing approach does not improve FORTE's speed relative to EITHER in propositional domains. This is due to a difference in revision strategy between the two systems. EITHER limits the explosiveness of its global search by only considering revisions at the lowest level of a theory. It only moves upward in the theory if revision at the lower level fails. FORTE, by contrast, always explores revision points at all levels in the theory. However, FORTE's hill-climbing approach does allow it to work with reasonable efficiency in first-order domains, where a global search algorithm would likely be overwhelmed by the explosion in the theory space.

FORTE's use of a hill-climbing approach leaves it vulnerable to local maxima. FORTE can usually escape local maxima within a single clause by using relational pathfinding or by deleting multiple antecedents. However, when the only way to escape a local maximum is to make simultaneous changes in two or more clauses, FORTE cannot escape. As discussed in Chapter 6, this has not been a significant problem.

10.2 First-order Learning

A closely related first-order system is FOCL, presented in [Pazzani, Brunk, and Silverstein, 1991]. FOCL is an extension of FOIL that allows the use of an initial theory to guide an inductive learning process. Clauses and portions of clauses from the input theory are considered for addition to the rules under development by FOIL. If adding antecedents from the input theory provides more information gain than adding a newly created antecedent, the terms from the theory are chosen. Thus, providing a good input theory provides a substantial boost to the learning process. This is not true theory revision, however, and much of the information contained in the initial theory can be lost.

FOCL's approach is also at a disadvantage when the initial theory is missing lower-level rules (see Chapter 7 and [Cohen, 1991]). Furthermore, since the initial theory is only used to aid an otherwise inductive process, the structure of the initial theory is lost. FORTE preserves as much of the initial theory as possible, and can modify rules at any level in a theory with equal effectiveness. FOCL has been used as the basis for an interactive theory revision system, KR-FOCL [Pazzani and Brunk, 1991]. KR-FOCL examines a trace of FOCL execution on a knowledge base (FOCL is not actually allowed to modify the original knowledge base), and uses the information to suggest revisions to the user. This is intended to detect and correct simple errors in a multi-level theory.

A different approach to first-order induction is presented in [Kijsirikul, Numao, and Shimura, 1991] in their system CHAM. CHAM performs first-order induction on the same types of problems as FOIL. However, CHAM uses mode[29] information together with a new heuristic, *likelihood*, that attempts to determine what relations are needed based on a similarity analysis of input and output variables. Unfortunately, the likelihood heuristic only applies to recursively structured variables such as lists, and even then only applies to a select set of problems. For example, there is no *a priori* similarity among the variables in the top-level clause of merge-sort.

In [Cohen, 1992], Cohen describes GRENDEL, the most versatile and arguable the most powerful first-order induction system available. GRENDEL is based on FOIL, but Cohen allows the user to give GRENDEL a meta-level program which guides the learning process. Using this method he has been able to closely simulate a number of different first-order inductive learning systems. Cohen's thesis is that any learning algorithm inevitably brings with it a learning bias, and by allowing the user to "program" GRENDEL, this bias is made explicit. However, GRENDEL is still inherently an inductive learner; initial theories can be introduced only in the sense that they can be explicitly coded as part of the meta-program that guides GRENDEL's learning.

[29]Mode information refers to the fixed use of particular variables for input or output. For example, a sort predicate might have the defined mode sort(+, -), meaning that the first variable is the input variable and the second variable is the output variable. Modes are especially beneficial whenever a logic program is written procedurally.

AUDREY, described in [Wogulis, 1991] is a simple first-order theory revision system. It takes a two-pass approach to revising theories. First, it specializes the theory to exclude all negative instances. Specialization is coarse; it deletes entire clauses from the theory until all negatives have become unprovable. Second, AUDREY generalizes the theory to include all positives. Generalization uses an abductive process, but only a single assumption may be made for any positive instance; generalization fails if more than one assumption is required.

A technique similar to relational pathfinding was developed independently in [Langley, 1980], in the context of learning production rules. Langley's technique differs from relational pathfinding in a number of ways, although the underlying idea of graph search is present in both. For example, Langley's search method is unidirectional, and less tightly constrained. On the other hand, his bias in accepting paths is more restrictive—Langley seeks paths which lead to unique productions, which essentially means that the path must contain all antecedents necessary to complete the production. FORTE, on the other hand accepts any path which helps discriminate between positives and negatives, even if further specialization may be necessary.

10.3 Logic Program Synthesis

The specification-based approaches to logic program synthesis cannot be directly compared to FORTE, since the underlying assumptions are very different. However, we can contrast the approaches at a high level. Using a specification presupposes that the user understands the problem well enough to provide a complete and correct specification. Given this, and the relatively large effort required to correctly formulate the specification, we are rewarded with a program that is guaranteed to be correct. The inductive approach followed by FORTE is more suited to problems where the user is not able to formulate a complete and correct specification. What knowledge the user does have goes into the relatively easy task of constructing input-output examples. If the user has some algorithmic knowledge of the problem, this can go into an initial theory, which gives FORTE a starting point in its task. However, the price we pay for these less stringent input requirements is that, while the resulting program will work correctly on the given instances, it may still not be the intended program.

Similarly, it is difficult to directly compare FORTE to the work of Shapiro [Shapiro, 1983]. Shapiro's PDS6 can debug much more complex programs than FORTE can, but it does so by asking the user to identify the correctness of predicate calls, to determine which clause in a predicate should be changed, and to actually write missing program clauses. Forte, on the other hand, determines where errors lie in the theory and develops corrections without any user interaction. When comparing FORTE to Shapiro's work (see Section 7.5), it is no surprise that the highly interactive PDS6 debug more complex errors than a fully automatic system.

Flener's work [Flener, 1991] is a closer analogy. He seeks to learn logic programs from positive input-output tuples supplemented with informal *properties* that provide information equivalent to what FORTE gains from the use of negative examples. He does not provide for the possibility of an initial program. Unfortunately, Flener presents no results to demonstrate the capabilities of his approach. His methods seem likely to be more effective than FORTE at developing good recursions, since he uses predefined program schemata, but he provides less powerful operators overall, and is thus likely to be less effective on nonrecursive learning problems.

10.4 Qualitative Model Building

In qualitative model building, [Coiera, 1989] presents GENMODEL, a method for inducing a qualitative model from qualitative behaviors. His approach is limited by the fact that behaviors must be completely specified, and his output models contain many mathematically redundant constraints due to the absence of dimensional analysis.[30] A more powerful system, MISQ, was developed independently in [Kraan, Richards, and Kuipers, 1991] and [Richards, Kraan, and Kuipers, 1992]. MISQ adds dimensional analysis to the model-building process, and is also able to work with incomplete behavioral information. The core learning algorithm of both GENMODEL and MISQ is similar to the version-space computation of most specific conjunctive hypotheses ([Mitchell, 1982]). FORTE's processing of most-specific theories subsumes this method, and thus FORTE can be used to implement MISQ. Relational pathfinding allows FORTE to induce models even when the user has omitted essential system variables, something that MISQ as described in [Kraan, Richards and Kuipers, 1991] cannot do. FORTE is also able to take an initial model as input, and revise that model to reflect observed behaviors.

GOLEM has also been applied to the problem of learning qualitative models [Bratko, Muggleton, and Varsek, 1991]. However, their method requires hand-generated negative information (i.e., examples of behaviors that the system does *not* exhibit), it does not completely implement the QSIM constraints (e.g., corresponding values are ignored), and it does not use dimensional information. GOLEM also requires extensionally defined background knowledge, whereas FORTE's fundamental domain theory allows background knowledge to be defined intensionally.

Although not a qualitative system, BACON.3 [Langley, 1977] performs the related task of building algebraic expressions. Since BACON works from numeric data to construct numeric equations, its operations are more tightly

[30]If a model is interpreted as a real physical system, then these constraints are not merely redundant, but actually incorrect. One would never, for example, add velocity and mass, even if doing so were plausible from the observed values.

constrained than those of MISQ. As a result, it is able to create equations that include many implicit intermediate variables (e.g., the equation $a(b+c) = d$ contains an implicit intermediate variable holding the value of $b + c$). However, working with numeric equations restricts BACON to relatively simple physical systems, since its library of functions must include the exact functions required to model the system.

Chapter 11
FUTURE WORK

11.1 ENHANCEMENTS TO FORTE

There are a number of possibilities for direct enhancements to FORTE. These are outlined in the following paragraphs.

11.1.1 Efficiency

FORTE's operator-based approach is highly effective, but also highly inefficient. One area with the potential to greatly improve FORTE's efficiency is the area of operator selection. If we could classify theory errors to determine which operator is most likely to provide a good revision for a particular type of error, we could eliminate a great deal of redundant work by selecting the proper operator first, and only resorting to other operators if the selected operator fails to develop a useful revision.

Another difficult but potentially worthwhile improvement would be the use of an ATMS [deKleer, 1986] to cache proof results. Currently, when developing possible revisions, FORTE tries to prove every instance many times over. While each proof uses a different theory, most subgoals remain the same. While using an ATMS imposes a substantial overhead, it is likely that the savings resulting from caching unchanged subgoals would provide a marked increase in speed.

11.1.2 Mode Information

Our philosophy in developing FORTE has been to avoid mode information. The rationale for this is that pure logic programs should have a purely declarative meaning, whereas mode dependencies imply a procedural interpretation. However, mode information can be a powerful aid in developing recursive predicates, and can limit the difficulty of writing the fundamental domain theory (since, without modes, predicates in the fundamental domain theory must be defined for all modes). Thus, a reasonable way to enhance FORTE's capabilities would be to allow the user the option of specifying mode information, and having FORTE make use of this information when it is available.

Using mode information would allow FORTE to prune its search space by rejecting antecedents that would leave required inputs uninstantiated. It would also make FORTE aware of antecedent ordering within clauses. Currently, since training instances are ground, FORTE is unaware of the way that a predicate will be called in practice. For example, FORTE produces the clause

```
merge_sort(A, B) :-
        split(A, C, D),
        merge(E, F, B),
        merge_sort(C, E),
        merge_sort(D, F).
```

If this clause were used as part of a sort program it would be highly inefficient, since merge will generate possible permutations of the output list until it finds the ones that correspond to the following merge_sort antecedents. The use of modes would eliminate this problem.

11.1.3 Negation

Negation does not increase the expressive power of logic programs—any predicate can be written without using negation. However, doing so can be inconvenient, and can increase the complexity of a theory. Hence, another possible enhancement would be to allow FORTE to add negations of literals. The simplest way to do this is, when specializing a clause, to consider the negation of all antecedents we currently consider, thus roughly doubling the number of possible antecedents. However, it is not clear how one would deal with the problem of floundering[31], other than simply disallowing non-ground negations of any literal that cannot be proven to be ground at that point in the clause. [Bain, 1991] presents an interesting approach to negation using CIGOL, which could be extended to any first-order learning system.

11.1.4 Deep Recursion

Learning recursive predicates is perhaps the single most challenging aspect of first-order theory revision. FORTE can induce or revise recursive predicates for which a training set is provided, and has limited capabilities to revise lower-level recursive predicates. However, giving FORTE more detailed knowledge of recursion schemes would be very helpful. We do not mean giving FORTE a fixed set of recursion templates, which would subsequently limit the types of recursive predicates it would be able to learn. However, we could explicitly identify the variable on which a predicate will recurse, and ensure that all base cases and recursive cases use this variable appropriately. We could also add knowledge about what it means for a recursion to be well-founded.

Deeply recursive theories pose a larger challenge, and it is difficult to see how to lift the current limitations. However, this is an area where a

[31]The problem of floundering arises in SLDNF resolution when a negative literal contains an uninstantiated variable. Consider the logic program

```
p(b).
q :- not(p(X)), X = a.
```

Read declaratively, the query q should succeed. However, since X is uninstantiated when the negative literal is resolved, the program fails.

merger between the two approaches to logic program synthesis may be valuable. Both techniques, inductive learning from input-output examples and derivation from specifications, find nested recursions to be challenging. However, each method can provide different information to aid the selection of appropriate recursion schemes.

11.1.5 Predicate Invention

FORTE currently does not invent new predicates. Predicate invention is unnecessary when learning in non-recursive domains, although introducing new predicates may simplify the theory.[32] However, when learning a concept that is expressed as nested recursions, predicate invention is essential. For example, a program for naive reverse cannot be written without using a lower-level recursive predicate like append.

Inverse resolution offers one approach to predicate invention. For example, the operators interconstruction and intraconstruction can be used to invent new predicates, although they have traditionally been used only to help produce a more compact theory. Relevant work in inverse resolution includes [Muggleton, 1987, [Muggleton and Buntine, 1988], and [Muggleton, 1990]. An abductive approach to predicate invention is outlined in [Wirth and O'Rorke, 1991].

11.1.6 Probabilistic Theories

Theories for many domains are best cast in terms of probabilistic reasoning, fuzzy logic, certainty factors, or other numerically-based techniques. While FORTE currently makes no provision for such techniques,

[32]To see this, consider an arbitrary nonrecursive multi-level theory. Each antecedent referencing a lower-level predicate can be partially evaluated (possible in several ways) by resolving it with lower-level clauses. Collecting all possible partial evaluations yields an equivalent single-level theory. Since the theory is nonrecursive, the set of partial evaluations is finite.

the field of reasoning under uncertainty has received a great deal of work. Many of the techniques could be combined with FORTE to produce a first-order theory revision system capable of reasoning in these domains. One approach to revision of uncertain theories appears in [Mahoney and Mooney, 1992].

11.1.7 Other Revision Operators

FORTE depends on having a variety of revision operators to help it avoid local maxima. Hence, one direction for future work would be the development of new types of operators. For example, Forte currently has no operator that simultaneously revises multiple clauses. For example, suppose that a theory is overly specialized. It may be necessary do delete multiple antecedents in order to improve the accuracy of the theory. However, if these antecedents are distributed over different clauses, FORTE will be unable to develop an appropriate revision, and will resort to learning new rules—essentially discarding whatever information is present in the existing overly-specialized rules. This is the sort of interaction that prevents FORTE from correctly debugging the PDS6 example in Section 7.5.

11.2 FUTURE WORK IN LOGIC PROGRAMMING

While many of the enhancements discussed in Section 11.1 would improve FORTE's performance in the domain of logic programming, there are two specific areas of work that deserve separate consideration.

11.2.1 Speed-up Learning

A new area of first-order learning deals not with correcting a theory but with making its execution more efficient. The goal of the learner is to identify preconditions that must be satisfied before rules are tried, thus preventing needless backtracking. The preconditions can be expressed either as control rules in a meta-program or simply added as the first antecedents to be evaluated by the target rules. Positive and negative instances are derived by monitoring normal program execution; when a rule succeeds during a successful proof a positive instance is recorded. When some other rule participates in the final proof a negative instance is recorded. From these instances, normal first-order induction derives the control rules. However, if approximate control rules already exist, first-order theory revision techniques could be applied to revise them. This approach is explored in [Zelle and Mooney, 1992].

11.2.2 Automated Tutoring

FORTE can correct bugs in an incorrect logic programs in a way that preserves much of the original structure. In essence, FORTE tried to revise a program to be correct, but along the lines the author intended rather than according to some model solution. This give FORTE the potential to be used as part of an automated tutoring system for logic programming. The instructor could prepare a training set for FORTE to go along with each programming assignment given to the class. Students who needed help understanding what was wrong with their programs could use FORTE to debug them. If the program is too complex for FORTE to debug, due perhaps to nested recursions, the student could provide explicit examples for the lower-level predicates.

Used in this way, FORTE would provide a learning tool to help students learn to write and debug logic programs. FORTE would provide an individualized critique of each student's program, suggesting only those changes necessary to make the program work—even if it the result differs from the model solution intended by the instructor. This after-the-fact evaluation allows the student to attempt an assignment and then see a correction, as opposed to the highly interactive approach required by [Shapiro, 1983].

FORTE and PDS6 are at opposite ends of a spectrum; FORTE is completely automatic and PDS6 is highly interactive. It would be possible to develop an intermediate approach, where the system would work automatically, but would ask the user for guidance on questionable actions. For example, the user might be asked to confirm the need for predicate invention, or perhaps to verify the correctness of a recursion scheme.

11.3 FUTURE WORK IN QUALITATIVE MODELLING

FORTE's application to qualitative modelling gives MISQ the capability to invent new system variables, as described in Chapter 8 and in [Richards, Kraan, and Kuipers, 1992]. This is an important contribution to the task of automatically deriving models from system behaviors. However, much of FORTE's power is currently unused on this problem, since theories in this domain consist of only a single clause. The next step forward in this area will be the ability to work with model transitions—to simultaneously maintain more than one system model, and to switch between them as needed.

In addition, prior implementations of MISQ were able to work directly with quantitative behaviors. This is an important capability for many reasons, but perhaps the most important is that it allows the system to work directly with sensor data—eliminating any need for a person to translate the data. There are two ways to bring quantitative processing together with FORTE's theory revision capabilities. One is to implement a new special-purpose version of MISQ that incorporates relational pathfinding and whatever other features of FORTE that will be useful in addressing the model transition task. The other is to add the quantitative pre-processing component of earlier versions of MISQ and use it as an example translator for FORTE.

Finally, a shortcoming of all versions of MISQ to date, including the FORTE implementation, is that MISQ includes redundant constraints in its models. While these constraints do not alter the model semantics, they can make large models difficult to understand. Many of these redundant constraints could be eliminated via subsumption analysis.

Chapter 12
CONCLUSIONS

This research has demonstrated the feasibility of general purpose first-order theory revision by developing a system, FORTE, which has been used to revise and induce theories in a wide range of domains. FORTE is an operator-based revision system, and includes a variety of powerful theory revision operators. Some of these were derived from prior work in the field, while others are new. Relational pathfinding, in particular, has proven to be a valuable revision operator in almost all of the domains FORTE has been tested on. Although FORTE uses hill-climbing techniques, during the more than 1000 test runs needed to generate the learning curves in Section 6.1 Forte was trapped by local maxima less than 1% of the time.

A first-order theory revision system is a significant advance over propositional theory revision systems. While many domains can be encoded as propositional learning problems, doing so often greatly increases their size and reduces their understandability. Moreover, many domains cannot be so encoded; for example, one cannot express useful programs in propositional logic. Thus, working in first-order logic opens many new fields to machine learning.

FORTE also provides a research contribution to two of the individual domains in which it has been tested. In logic programming, FORTE has been able to automatically debug incorrect programs that previously would have required an oracle-based system. Its ability to revise programs while preserving their basic structure represents the beginning of an effective automatic tutoring system for logic programming. In one test, FORTE was able to revise multiple bugs in a large logic program which performs propositional inductive learning. Extensions to FORTE could lead to a highly effective automated tutoring system.

In the domain of qualitative modelling, FORTE can induce models from examples of system behavior, just as the special purpose system MISQ is able to do. However, FORTE's relational pathfinding algorithm provides the ability to create system variables as needed, in order to produce a correct model. This is an important capability, since a user trying to understand the operation of a system cannot be expected to know in advance which variables will be necessary to model that system. Theory revision capabilities translate, in this domain, to an ability to perform model revision and diagnosis. These capabilities were demonstrated on the space shuttle reaction control system. Future work in this area may lead to the ability to induce, revise, and diagnose systems of qualitative models connected by transitions.

These domains and many others currently rely on humans to perform knowledge acquisition and theory revision. Knowledge acquisition is an expensive and time-consuming task. Yet, once obtained, knowledge requires maintenance for the same reasons any software does: undetected errors, a changing environment, or extensions to unplanned areas of application. Automating theory revision will allow modifications to be made to a knowledge base without recalling the knowledge engineer and domain expert. This, in turn, will lead to lower costs and more reliable knowledge bases.

Appendix A
HINTON'S FAMILY DOMAIN

The Hinton data set of two isomorphically identical families, one of which is shown in Figure 44. Positive instances consist of all correct instances of the twelve concepts: husband/2, wife/2, father/2, mother/2, son/2, daughter/2, brother/2, sister/2, uncle/2, aunt/2, nephew/2, and niece/2. Negative instances are "near-misses," i.e., instances which are provable by a theory which differs by only one or two antecedents from a correct theory. For example, negative instances for uncle/2 include examples of aunt/2 and grandfather/2.

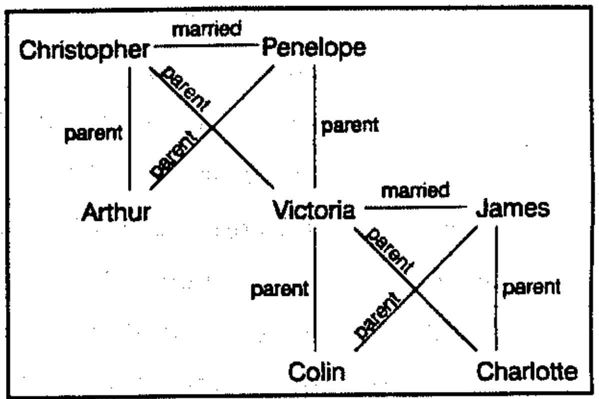

Figure 44. One of the families in Hinton's data.

The fundamental domain theory for this data set adds only one piece of knowledge: that marriage is commutative. The definition of marriage in the fundamental domain theory is

```
married(X, Y) :- example(married(X, Y)), !.
married(X, Y) :- example(married(Y, X)).
```

A correct theory for the concepts in this domain appears in Figure 45.

```
wife(X, Y) :- gender(X, female), married(X, Y).
husband(X, Y) :- gender(X, male), married(X, Y).
mother(X, Y) :- gender(X, female), parent(X, Y).
father(X, Y) :- gender(X, male), parent(X, Y).
daughter(X, Y) :- gender(X, female), parent(Y, X).
son(X, Y) :- gender(X, male), parent(Y, X).
sister(X, Y) :- gender(X, female), sibling(X,Y).
brother(X, Y) :- gender(X, male), sibling(X,Y).
aunt(X, Y) :- gender(X, female), au(X, Y).
uncle(X, Y) :- gender(X, male), au(X, Y).
niece(X, Y) :- gender(X, female), au(Y, X).
nephew(X, Y) :- gender(X, male), au(Y, X).
au(X, Y) :- sibling(X, B), parent(B, Y).
au(X, Y) :- married(X, A), sibling(A, C), parent(C, Y).
sibling(X, Y) :- parent(A, X), parent(A, Y), X \= Y.
```

Figure 45. Correct theory for family relationships.

Appendix B
LARGE FAMILY DOMAIN

This appendix describes the data set for the family domain and the randomly corrupted theories used for revision, and shows a sample FORTE execution in this domain.

B.1 DATA SET

The data set describes a large family consisting of 86 people in five generations (see Figure 46). The data set lacks the artificial regularity of Hinton's data. Positive instances consist of all correct instances of the same twelve relations that were used with Hinton's data. Negative instances were randomly generated; there are twice as many negatives as positives.

B.2 THEORIES

Since the same concepts are defined both here and for Hinton's data, the theory in Figure 45 is also correct for this domain. The five randomly corrupted theories used to generate the revision learning curve in Chapter 5 appear below.

```
wife(X,Y) :- gender(X,female), married(X,Y).
husband(X,Y) :- gender(X,male).
mother(X,Y) :- gender(X,female), parent(X,Y).
father(X,Y) :- gender(X,male), parent(X,Y).
daughter(X,Y) :- gender(X,female), parent(Y,X).
son(X,Y) :- gender(X,male), parent(Y,X).

sister(X,Y) :- gender(X,female), sibling(X,Y).
brother(X,Y) :- gender(X,male), au(Z,Y).
aunt(X,Y) :- gender(X,female), au(X,Y).
uncle(X,Y) :- gender(X,male), au(X,Y).
niece(X,Y) :- gender(X,female), au(Y,X).
```

```
wife(X,Y) :- gender(X,female), married(X,Y).
husband(X,Y) :-gender(X,male), married(X,Y).
mother(X,Y) :- married(Z,Y).
father(X,Y) :- gender(X,male), parent(X,Y).
daughter(X,Y) :- gender(X,female), parent(Y,X).
son(X,Y) :- gender(X,male), parent(Y,X),
        married(Z,X).
sister(X,Y) :- gender(X,female), sibling(X,Y).
brother(X,Y) :- gender(X,male), sibling(X,Y).

uncle(Z,Y) :- gender(X,male), au(X,Y).
niece(X,Y) :- gender(X,female), au(Y,X).
nephew(X,Y) :- gender(X,male), au(Y,X).
```

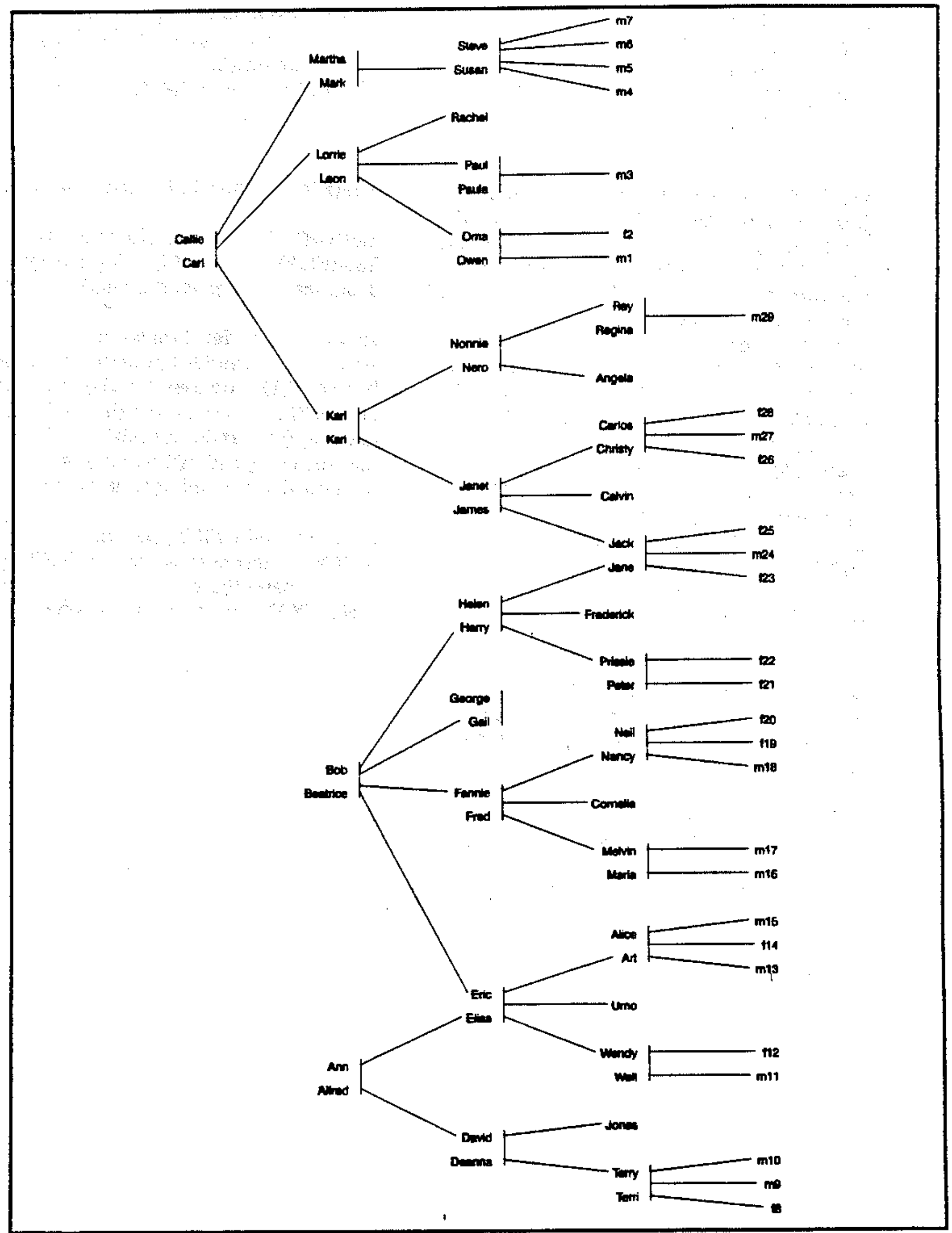

Figure 46. Graphical representation of the large family domain.

```
au(X,Y) :- sibling(X,B), parent(B,Y), married(B,Z).
au(X,Y) :- married(X,A), sibling(A,C),
         parent(C,Y), sibling(X, Z).
sibling(X,Y) :- parent(A,X), parent(A,Y),
         X \= Y.

wife(X,Y) :- gender(X,female), married(X,Y).
husband(X,Y) :- gender(X,male), married(X,Y).
mother(X,Y) :- gender(X,female), parent(X,Y).
father(X,Y) :- gender(X,male), parent(X,Y).
daughter(X,Y) :- gender(X,female), parent(Y,X).
daughter(X,Y) :- married(X,Z).
son(X,Y) :- gender(X,female), parent(Y,X).
sister(X,Y) :- gender(X,female), sibling(X,Y).
brother(X,Y) :- gender(X,male), sibling(X,Y).
aunt(X,Y) :- gender(X,female), au(X,Y).
uncle(X,Y) :- gender(X,male), au(X,Y).
niece(X,Y) :- gender(X,female), au(Y,X).
nephew(X,Y) :- gender(X,male), au(Y,X).
nephew(X, Y) :- au(Y, Z), X \= Y.
au(X,Y) :- sibling(X,B), parent(B,Y).
au(X,Y) :- married(X,A), sibling(A,C),
                   parent(C,Y).
sibling(X,Y) :- parent(A,X), parent(A,Y),
         X \= Y.

wife(X,Y) :- gender(X,female), married(X,Y).
husband(X,Y) :- gender(X,male), married(X,Y).
mother(X,Y) :- gender(X,female), parent(X,Y).
father(X,Y) :- gender(X,male), parent(X,Y).
daughter(X,Y) :- gender(X,female), parent(Y,X).
son(X,Y) :- gender(X,male), parent(Y,X).
sister(X,Y) :- gender(X,female), sibling(X,Y).
brother(X,Y) :- gender(X,male), sibling(X,Y).
aunt(X,Y) :- gender(X,female), au(X,Y).
uncle(X,Y) :- gender(X,male), au(X,Y), parent(X,Y).
uncle(X,Y) :- married(X,A), sibling(A,B).
niece(X,Y) :- gender(X,female), au(Y,X).
nephew(X,Y) :- gender(X,male), au(Y,X).
au(X,Y) :- sibling(X,B), parent(B,Y).
au(X,Y) :- married(X,A), sibling(A,C), parent(C,Y).
sibling(X,Y) :- parent(A,X), parent(A,Y), X \= Y.
```

```
au(X,Y) :- sibling(X,B), parent(B,Y).
au(X,Y) :- married(X,A), sibling(A,C),
         parent(C, Y).
sibling(X,Y) :- parent(A,X), parent(A,Y),
         X \= Y.

wife(X,Y) :- gender(X,female), married(X,Y).

mother(X,Y) :- gender(X,female), parent(X,Y).
father(X,Y) :- gender(X,male), parent(X,Y).
daughter(X,Y) :- gender(X,female), parent(Y,X).

son(X,Y) :- gender(X,male), parent(Y,X).
sister(X,Y) :- gender(X,female), sibling(X,Y).
brother(X,Y) :- gender(X,male), sibling(X,Y).
aunt(X,Y) :- gender(X,female), au(X,Y).
uncle(X,Y) :- gender(X,male), au(X,Y).
niece(X,Y) :- gender(X,female), au(Y,X).
nephew(X,Y) :- gender(X,male), au(Y,X).

au(X,Y) :- sibling(X,B), parent(B,Y).
au(X,Y) :- married(X,A), sibling(A,C),
         parent(C,Y).
sibling(X,Y) :- parent(A,X), parent(A,Y),
         X \= Y.
```

B.3 SAMPLE EXECUTION

```
Forte version 2.15 -- Test parameters

Data set, initial theory, and domain knowledge:  ../Data/family, ../Data/fam2,../Domain/family
The data set contains 2232 instances.
```

```
Language bias is:
[depth_limit(5),use_attr,use_relations,use_theory,use_built_in,relation_tuning(highly_relational)]
Output detail set to 3, and initial random seed is random

Test series of 1 independent runs
Training set is 100 instances; test set is 2132 instances

Initial Theory

sibling(A,B):-parent(C,A),parent(C,B),A\=B.

au(A,B):-married(A,C),sibling(C,D),parent(D,B).
au(A,B):-sibling(A,C),parent(C,B).

nephew(A,B):-gender(A,male),au(B,A).

niece(A,B):-gender(A,female),au(B,A).

uncle(A,B):-gender(C,male),au(C,B).

brother(A,B):-gender(A,male),sibling(A,B).

sister(A,B):-gender(A,female),sibling(A,B).

son(A,B):-gender(A,male),parent(B,A),married(C,A).

daughter(A,B):-gender(A,female),parent(B,A).

father(A,B):-gender(A,male),parent(A,B).

mother(A,B):-married(C,B).

husband(A,B):-gender(A,male),married(A,B).

wife(A,B):-gender(A,female),married(A,B).

---------- Beginning test run with seed:  random(29919,10151,10674,425005073) ----------

Initial train/test accuracy:  0.88, 0.850375

Selected revision by add_rule scores (6, -2) by replacing the old
aunt(A,B):-fail.
with the new
aunt(B,C):-au(B,C).
aunt(A,B):-fail.
end revision

Selected revision by delete_antecedent scores (2, 1) by replacing the old
son(A,B):-gender(A,male),parent(B,A),married(C,A).
with the new
son(A,B):-gender(A,male),parent(B,A).
end revision

Selected revision by delete_rule scores (2, 0) by replacing the old
mother(A,B):-married(C,B).
```

```
with the new
mother(A,B):-fail.
end revision

Selected revision by add_antecedent scores (2, -1) by replacing the old
uncle(A,B):-gender(C,male),au(C,B).
with the new
uncle(A,B):-gender(C,male),au(C,B),C=A.
end revision

Revised Theory

aunt(A,B):-au(A,B).

sibling(A,B):-parent(C,A),parent(C,B),A\=B.

au(A,B):-married(A,C),sibling(C,D),parent(D,B).
au(A,B):-sibling(A,C),parent(C,B).

nephew(A,B):-gender(A,male),au(B,A).

niece(A,B):-gender(A,female),au(B,A).

uncle(A,B):-gender(A,male),au(A,B).

brother(A,B):-gender(A,male),sibling(A,B).

sister(A,B):-gender(A,female),sibling(A,B).

son(A,B):-gender(A,male),parent(B,A).

daughter(A,B):-gender(A,female),parent(B,A).

father(A,B):-gender(A,male),parent(A,B).

mother(A,B):-fail.

husband(A,B):-gender(A,male),married(A,B).

wife(A,B):-gender(A,female),married(A,B).

            Initial   Initial   Initial               Final     Final     Final
Training     Theory  Training  Test Set  Training    Theory  Training  Test Set
Set Size       Size  Accuracy  Accuracy      Time      Size  Accuracy  Accuracy
--------   --------  --------  --------  --------  --------  --------  --------
     100         44     88.00     85.04      1992        45    100.00     97.19

---------- End of test run ----------
```

Appendix C
KRK ILLEGALITY DATA

The king-rook-king domain consists of chess positions with a white rook and king and a black king, with white to move. A position is positive if it is illegal. A position is illegal if the black king is in check by the white rook, if the two kings are adjacent, or if two pieces occupy the same square.

C.1 DATA SET

The data set was randomly generated, and contains 684 positives and 1316 negatives. The fundamental domain theory provides definitions for row/column adjacency (numerical difference of 1), less-than, and equality.

C.2 THEORIES

The five randomly corrupted versions of the multi-level theory appear below, along with one sample revision of each theory using 50 training instances.

Corrupted theory 1

```
krk(WKR, WKF, WRR, WRF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, New) :- same_square(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WRR, WRF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- adjacent_squares(WKR, WKF, BKR, BKF).
krk(WKR, WKF, Rank, WRF, Rank, BKF) :- line_attack(WKR, Rank, WKF, WRF, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- line_attack(WKF, WKR, WRR, BKR, BKF).
krk(WKR, WKF, WRR, File, BKR, File) :- line_attack(WKF, File, WKR, WRR, BKR).

same_square(A, B, A, B).

adjacent_squares(Rank1, File1, Rank2, File2) :- adj(Rank1, Rank2), adj(File1, File2).
adjacent_squares(Rank1, File, Rank2, File) :- adj(Rank1, Rank2).
adjacent_squares(Rank, File1, Rank, File2) :- adj(File1, File2).
```

```
line_attack(WK, Others, A, B, C) :- not_equal(WK, Others).
line_attack(Same, Same, WK, WR, BK) :- less(WK, WR), less(WK, BK).
line_attack(Same, Same, WK, WR, BK) :- less(WR, WK), less(BK, WK).
```

Revised theory 1

```
krk(A,B,C,D,E,F):-same_square(A,B,E,G),equal(G,F).
krk(A,B,C,D,E,F):-line_attack(B,A,C,E,F),equal(F,E),less(D,C).
krk(A,B,C,D,E,F):-line_attack(B,A,C,E,F),adj(F,F).
krk(A,B,C,D,E,D):-line_attack(B,D,A,C,E).
krk(A,B,C,D,C,E):-line_attack(A,C,B,D,E).
krk(A,B,C,D,E,F):-adjacent_squares(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(C,D,E,F).

same_square(A,B,A,B).

adjacent_squares(A,B,A,C):-adj(B,C).
adjacent_squares(A,B,C,B):-adj(A,C).
adjacent_squares(A,B,C,D):-adj(A,C),adj(B,D).

line_attack(A,A,B,C,D):-less(C,B),less(D,B).
line_attack(A,A,B,C,D):-less(B,C),less(B,D).
line_attack(A,B,C,D,E):-not_equal(A,B).
```

Corrupted theory 2

```
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, WRR, WRF).
krk(WKR, WKF, WRR, WRR, BKR, BKF) :- same_square(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WRR, WRF, BKR, BKF).
krk(WKR, WKF, Rank, WRF, Rank, BKF) :- line_attack(WKR, Rank, WKF, WRF, BKF).
krk(WKR, WKF, WRR, File, BKR, File) :- line_attack(WKF, File, WKR, WRR, BKR).

same_square(A, B, A, B).

adjacent_squares(Rank1, File1, Rank2, File2) :- adj(Rank1, Rank2), adj(File1, File2).
adjacent_squares(Rank1, File, Rank2, File) :- adj(Rank1, Rank2).
adjacent_squares(Rank, File1, Rank, File2) :- adj(File1, File2).

line_attack(WK, Others, A, B, C) :- not_equal(New, Others).
line_attack(Same, Same, WK, WR, BK).
line_attack(Same, Same, WK, WR, BK) :- not_less(WR, WK), less(BK, WK).
```

Revised theory 2

```
krk(A,B,C,D,E,F):-equal(F,D).
krk(A,B,C,D,E,F):-adjacent_squares(E,F,A,B).
krk(A,B,C,D,E,F):-equal(E,C).
krk(A,B,C,D,E,D):-line_attack(B,D,A,C,E).
krk(A,B,C,D,C,E):-line_attack(A,C,B,D,E).
krk(A,B,C,D,E,F):-same_square(C,D,E,F).
krk(A,B,C,C,D,E):-same_square(A,B,D,E).
krk(A,B,C,D,E,F):-same_square(A,B,C,D).

same_square(A,B,A,B).

adjacent_squares(A,B,A,C):-adj(B,C).
adjacent_squares(A,B,C,B):-adj(A,C).
adjacent_squares(A,B,C,D):-adj(A,C),adj(B,D).

line_attack(A,A,B,C,D):-not_less(C,B),less(D,B).
line_attack(A,A,B,C,D).
line_attack(A,B,C,D,E):-not_equal(F,B).
```

<u>Corrupted theory 3</u>

```
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, WRR, WRF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WRR, WRF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- adjacent_squares(WKR, WKF, BKR, BKF).
krk(WKR, WKF, Rank, WRF, Rank, BKF) :- line_attack(WKR, Rank, WKF, WRF, BKF).
krk(WKR, WKF, WRR, File, BKR, File) :- line_attack(WKF, File, WKR, WRR, BKR).

same_square(A, B, A, B).

adjacent_squares(Rank1, File1, Rank2, File2) :- adj(Rank1, Rank2), adj(File1, File2).
adjacent_squares(Rank1, File, Rank2, File) :- adj(Rank1, Rank2).
adjacent_squares(Rank, File1, Rank, File2) :- adj(File1, File2).

line_attack(WK, Others, A, B, C) :- not_equal(WK, Others).
line_attack(Same, Same, WK, WR, BK) :- less(WK, WR), less(WK, BK).
line_attack(Same, Same, WK, WR, BK) :- less(BK, WK), less(BK, WK).
```

<u>Revised theory 3</u>

```
krk(A,B,C,D,E,D):-line_attack(B,D,A,C,E).
krk(A,B,C,D,C,E):-line_attack(A,C,B,D,E).
krk(A,B,C,D,E,F):-adjacent_squares(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(C,D,E,F).
krk(A,B,C,D,E,F):-same_square(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(A,B,C,D).

same_square(A,B,A,B).

adjacent_squares(A,B,A,C):-adj(B,C).
adjacent_squares(A,B,C,B):-adj(A,C).
adjacent_squares(A,B,C,D):-adj(A,C),adj(B,D).

line_attack(A,A,B,C,D):-less(D,B).
line_attack(A,A,B,C,D):-less(B,C),less(B,D).
line_attack(A,B,C,D,E):-not_equal(A,B).
```

Currupted theory 4

```
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, WRR, WRF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- adjacent_squares(WKR, WKF, BKR, BKF).
krk(WKR, WKF, Rank, WRF, Rank, BKF) :- line_attack(WKR, Rank, WKF, WRF, BKF).
krk(WKR, WKF, WRR, File, BKR, File) :- line_attack(WKF, File, WKR, WRR, BKR).
krk(WKR, WKF, WRR, File, BKR, File) :- same_square(WRR, File, BKR, File),
                                        line_attack(BKR, File, WKR, WRR, BKR).

same_square(A, B, A, B).

adjacent_squares(Rank1, File1, Rank2, File2) :- adj(Rank1, Rank2), adj(File1, File2).
adjacent_squares(Rank1, File, Rank2, File) :- adj(Rank1, Rank2).
adjacent_squares(Rank, File1, Rank, File2) :- adj(File1, File2), adj(File1, Rank).

line_attack(WK, Others, A, B, C) :- not_equal(WK, Others).
line_attack(Same, Same, WK, WR, BK) :- less(WK, BK), less(WK, BK).
line_attack(Same, Same, WK, WR, BK) :- less(WR, WK), less(BK, WK).
```

Revised theory 4

```
krk(A,B,C,D,E,D):-same_square(C,D,E,D),line_attack(E,D,A,C,E).
krk(A,B,C,D,E,D):-line_attack(B,D,A,C,E).
krk(A,B,C,D,C,E):-line_attack(A,C,B,D,E).
krk(A,B,C,D,E,F):-adjacent_squares(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(A,B,C,D).

same_square(A,B,A,B).

adjacent_squares(A,B,A,C):-adj(B,C),adj(B,A).
adjacent_squares(A,B,C,B):-adj(A,C).
adjacent_squares(A,B,C,D):-adj(A,C),adj(B,D).

line_attack(A,A,B,C,D):-less(C,B),less(D,B).
line_attack(A,A,B,C,D):-less(B,D).
line_attack(A,B,C,D,E):-not_equal(A,B).
```

Currupted theory 5

```
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, WRR, WRF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- same_square(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- adjacent_squares(WKR, WKF, BKR, BKF).
krk(WKR, WKF, WRR, WRF, BKR, BKF) :- line_attack(WKR, BKR, WKF, WRF, BKF),
                                      line_attack(WKF, BKF, WKR, WRR, BKR).
krk(WKR, WKF, Rank, WRF, Rank, BKF) :- line_attack(WKR, Rank, WKF, WRF, BKF).
krk(WKR, WKF, WRR, File, BKR, File) :- line_attack(WKF, File, WKR, WRR, BKR).
```

```
same_square(A, B, A, B).

adjacent_squares(Rank1, File1, Rank2, File2) :- adj(Rank1, Rank2), adj(File1, File2).
adjacent_squares(Rank1, File, Rank2, File) :- adj(Rank1, Rank2).

line_attack(WK, Others, A, B, C) :- not_equal(WK, Others).
line_attack(Same, Same, WK, WR, BK) :- less(WK, WR), less(WK, BK).
line_attack(Same, Same, WK, WR, BK) :- less(WR, WK), less(BK, WK).
```

Revised theory 5

```
krk(A,B,C,D,E,D):-same_square(C,D,E,D),line_attack(E,D,A,C,E).
krk(A,B,C,D,E,D):-line_attack(B,D,A,C,E).
krk(A,B,C,D,C,E):-line_attack(A,C,B,D,E).
krk(A,B,C,D,E,F):-adjacent_squares(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(A,B,E,F).
krk(A,B,C,D,E,F):-same_square(A,B,C,D).

same_square(A,B,A,B).

adjacent_squares(A,B,A,C):-adj(B,C),adj(B,A).
adjacent_squares(A,B,C,B):-adj(A,C).
adjacent_squares(A,B,C,D):-adj(A,C),adj(B,D).

line_attack(A,A,B,C,D):-less(C,B),less(D,B).
line_attack(A,A,B,C,D):-less(B,D).
line_attack(A,B,C,D,E):-not_equal(A,B).
```

C.3 SAMPLE RUN

The sample run below is an inductive run (i.e., an empty initial theory) using a training set of 100 instances.

```
Forte version 2.15 -- Test parameters

Data set, initial theory, and domain knowledge:  ../Data/krk-flat, ../Data/empty,
                                                                      ../Domain/krk
The data set contains 2000 instances.
Language bias is:
[depth_limit(10),nonconjunctive,use_relations,use_theory,no_new_vars,
                                          relation_tuning(non_relational)]
Output detail set to 3, and initial random seed is random


Test series of 1 independent runs
Training set is 100 instances; test set is 1900 instances
```

Initial Theory

```
---------- Beginning test run with seed:  random(831,15285,7013,425005073)
----------

Initial train/test accuracy:  0.59, 0.661579

Revised Theory

krk(A,B,C,D,E,F):-equal(C,A),equal(D,B).
krk(A,B,C,D,E,F):-equal(E,A),adj(B,F).
krk(A,B,C,D,E,F):-not_less(E,A),adj(A,E),not_adj(D,B).
krk(A,B,C,D,E,F):-adj(E,A),adj(F,B).
krk(A,B,C,D,E,F):-equal(E,C).
krk(A,B,C,D,E,F):-equal(F,D).

                     Initial   Initial   Initial                Final     Final     Final
          Training    Theory  Training  Test Set  Training     Theory  Training  Test Set
          Set Size      Size  Accuracy  Accuracy      Time       Size  Accuracy  Accuracy
          --------  --------  --------  --------  --------   --------  --------  --------
               100         0     59.00     66.16       147         17    100.00     93.58

---------- End of test run ----------
```

Appendix D
DIRECTED PATH

This appendix shows the original incorrect student programs along with the corresponding correct programs generated by FORTE.

Incorrect theory	Correctly revised theory
path(A, B) :- edge(A, B). path(A, B) :- path(A, C), path(C, B).	path(A, B) :- edge(A, B). path(A, B) :- edge(C, B), path(A, C).
path(A, B) :- edge(A, B). path(A, B) :- edge(B, A). path(A, B) :- edge(A, C), edge(D, B), path(A, C).	path(A, B) :- edge(A, B). path(A, B) :- edge(C, B), path(A, C).
path(A, B) :- edge(A, B). path(A, B) :- edge(B, A). path(A, B) :- edge(A, C), edge(B, D), path(C, D).	path(A, B). path(A, B) :- edge(C, B), path(A, C).
path(A, B) :- edge(A, B). path(A, B) :- edge(A, C), edge(D, B), path(C, D).	path(A, B) :- edge(A, B). path(A, B) :- edge(A, C), edge(C, B). path(A, B) :- edge(A,C), edge(D, B), path(C, D).

Appendix E
INSERT-AFTER

This appendix includes the original incorrect student programs for the insert-after problem, along with the corresponding revised programs generated by FORTE.

Incorrect theory	Correctly revised theory

`insert_after([A|B], A, C, [A,C|B]).`
`insert_after([A|B], C, D, [A|E]) :-`
`        insert_after(B, C, D, E).`
	`insert_after([A|B], A, C, [A,C|B]).`
	`insert_after([A|B], C, D, [A|E]) :-`
	`        C \= A, insert_after(B, C, D, E).`

`insert_after([A|B], A, C, [A,C|B]).`
`insert_after([], A, B, []).`
`insert_after([A|B], C, D, [A|E]) :-`
`        insert_after(B, C, D, E).`
	`insert_after([A|B], A, C, [A,C|B]).`
	
	`insert_after([A|B], C, D, [A|E]) :-`
	`        C \= A, insert_after(B, C, D, E).`

`insert_after([A|B], A, C, [A,C|B]).`
`insert_after([A], A, B, [A,B]).`
`insert_after([A|B], C, D, [A|E]) :-`
`        insert_after(B, C, D, E).`
	`insert_after([A|B], A, C, [A,C|B]).`
	
	`insert_after([A|B], C, D, [A|E]) :-`
	`        C \= A, insert_after(B, C, D, E).`

`insert_after([], A, B, []).`
`insert_after([A|B], C, D, E) :-`
`        insert_after(B, C, D, [A|E]).`
	`insert_after([A|B], A, C, [A,C|B]).`
	`insert_after([A|B], C, D, [A|E]) :-`
	`        C \= A, insert_after(B, C, D, E).`

`insert_after([], A, B, []).`
`insert_after([A|B], C, D, [A|B]) :-`
`        insert_after(B, C, D, E).`
	`insert_after([A|B], A, C, [A,C|B]).`
	`insert_after([A|B], C, D, [A|E]) :-`
	`        A \= C, insert_after(B, C, D, E).`

`insert_after([A], A, B, [A,B|C]).`
`insert_after([A|B], C, D, [E|B]) :-`
`        insert_after(B, C, D, B).`
	`insert_after([A|B], A, C, [A,C|B]).`
	`insert_after([A|B], C, D, [A|E]) :-`
	`        C \= A, insert_after(B, C, D, E).`

`insert_after([A|B], C, D, [A|E]) :-`
`insert_after(B, C, D, E).`
`insert_after([A|B], A, C, [A,C|D]) :-`
`        insert_after(B, A, C, D).`
	`insert_after([A|B], A, C, [A,C|B]).`
	
	`insert_after([A|B], C, D, [A|E]) :-`
	`        C \= A, insert_after(B, C, D, E).`

```
insert_after([A|B], A, C, [A,C|B]).
insert_after([A|B], C, D, [E|F]) :-
        insert_after(B, C, D, F).
```

```
insert_after([A|B], C, D, [A|E]) :-
        insert_after(B, C, D, E).
```

```
insert_after([A|B], A, C, [A,C|B]).
insert_after([A|B], C, D, [A|E]) :-
        C \= A, insert_after(B, C, D, E).
```

```
insert_after([A|B], A, C, [A,C|B]).
insert_after([A|B], C, D, [A|E]) :-
        C \= A, insert_after(B, C, D, E).
```

Appendix F
MERGE SORT

This appendix shows the original incorrect student programs for the merge sort program, along with the corresponding corrected programs generated by FORTE.

Incorrect theory

```
merge_sort([A,B], [B,A]) :- A >= B.
merge_sort([A,B], [A,B]) :- A < B.
merge_sort(A, B) :-
        split(A, C, D),
        merge_sort(C, E), merge_sort(D, F),
        merge(E, F, B).


merge_sort([], []).
merge_sort([A], [B]).
merge_sort(A, B) :-
        split(A, C, D),
        merge_sort(C, E), merge_sort(D, F),
        merge(E, F, B).


merge_sort([], []).

merge_sort(A, B) :-
        split(A, C, D),
        merge_sort(C, E), merge_sort(D, F),
        merge(E, F, B).


merge_sort([],[]).
merge_sort([A|B], [A|C]):- merge_sort(B, C).
```

Correctly revised theory

```
merge_sort(A, B) :- merge(A, B, A).

merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).


merge_sort([], []).
merge_sort([A], [A]).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).


merge_sort([], []).
merge_sort([A], [A]).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).


merge_sort([],[]).
merge_sort([A|B], A) :- merge_sort(B, []).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).
merge_sort(A, A).
merge_sort(A, B) :-
        split(A, C, D),
        merge_sort(C, E), merge_sort(D, F),
        merge(B, E, F).
```

```
merge_sort([A], [A]).
merge_sort(A, B) :-
        split(A, C, D),
        merge_sort(C, E), merge_sort(D, F),
merge(E, F, B).
```

```
merge_sort(A, A).
merge_sort(A,B) :-
        split(A, C, D),
        merge(C, D, B),
        merge_sort(A, B).
```

```
merge_sort(A, B) :-
        split(A, C, D), merge(C, D, B).
```

```
merge_sort(A, B) :-
        split(A, C, D), merge(C, D, E),
        merge_sort(E, B).
```

```
merge_sort([], []).
merge_sort([A], [A]).
merge_sort([A,B|C], D) :-
        split([A,B|C], E, F),
        merge_sort(E, G),  merge_sort(F, H),
        merge(G, H, D).
```

```
merge_sort(A, B) :- merge(A, B, A).
merge_sort(A, B) :-
        merge(C, D, B), split(A, E, F),
        merge_sort(E, C), merge_sort(F, D).
```

```
merge_sort([], A) :- merge([], [], A).
```

```
merge_sort([A], [A]).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).
```

```
merge_sort(A, B) :- merge(A, B, A).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).
```

```
merge_sort(A, B) :- merge(A, B, A).
merge_sort(A, B) :-
        merge(C, D, B), split(A, E, F),
        merge_sort(E, C), merge_sort(F, D).
```

```
merge_sort([], []).
merge_sort(A, A) :-
        merge(A, A, B), split(B, A, A).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).
```

```
merge_sort([], []).
merge_sort([A], [A]).
merge_sort(A, B) :-
        split(A, C, D), merge(E, F, B),
        merge_sort(C, E), merge_sort(D, F).
```

Appendix G
DECISION-TREE INDUCTION

This appendix shows the decision-tree induction program, followed by a sample revision run on a buggy version of this program.

G.1 DECISION-TREE PROGRAM

The top three predicates (induce_tree/3, induce_tree/5, and choose_attribute/5) were included in the initial theory, with the remainder of the program placed in the fundamental domain theory.

```
/* Top-level: given attributes and examples, produce decision tree */
induce_tree(Attr, Examples, Tree) :-
        induce_tree(A1, [], Attr, Examples, Tree).

/* Given no examples, produce an empty tree */
induce_tree(_, [], _, [], [] ).

/* If examples are pure, the tree is the class of the examples */
induce_tree( A1, [], A2, Examples, Class ) :-
        Examples = [Example|_],
        pure(Examples),
        Example = [Class, _].

/* If the examples are impure, choose best attribute as root of subtree */
induce_tree( _, [], Attributes, Examples, [Attr_name, Subtrees] ) :-
        Examples = [_|_],
        impure(Examples),
        choose_attribute( Attributes, Examples, 0, _, Attribute),
        fdt_delete( Attribute, Attributes, Rest_atts),
        Attribute = [ Attr_name, Values ],
        induce_tree( Attr_name, Values, Rest_atts, Examples, Subtrees).

/* create subtree branch for each value in selected attribute */
induce_tree(Name, [Val|Vals], Rest_atts, Examples, [[Val, Tree] | Trees]) :-
        Vals = [_|_],
        attval_subset(Name, Val, Examples, Example_subset),
        induce_tree(A1, [], Rest_atts, Example_subset, Tree),
        induce_tree(Name, Vals, Rest_atts, Examples, Trees).

/* last subtree branch */
induce_tree(Name, [Val], Rest_atts, Examples, [[Val, Tree]]) :-
        attval_subset(Name, Val, Examples, Example_subset),
        induce_tree(A1, [], Rest_atts, Example_subset, Tree).

/* choose attribute yielding highest purity */
choose_attribute( [], _, _, Best, Best).
choose_attribute( [Attr|Attrs], Examples, Best_purity, Best_so_far, Best) :-
        purity(Attr, Examples, Purity),
        less(Best_purity, Purity),
        choose_attribute( Attrs, Examples, Purity, Attr, Best ).
choose_attribute( [Attr|Attrs], Examples, Best_purity, Best_so_far, Best) :-
        purity(Attr, Examples, Purity),
        less_or_equal(Purity, Best_purity),
        choose_attribute( Attrs, Examples, Best_purity, Best_so_far, Best ).

/* determine average purity of leaves, if split on this attribute */
purity(Attr, Examples, Purity) :-
        Attr = [Name, Values],
        get_purities(Name, Values, Examples, Purities),
        average_purities(Purities, Purity).

get_purities(_, [], _, []).
get_purities(Name, [Value|Values], Examples, [Purity|Purities]) :-
        attval_subset(Name, Value, Examples, Example_subset),
        calc_purity(Example_subset, Purity),
        get_purities(Name, Values, Examples, Purities).

calc_purity(Examples, Purity) :-
        count_positives(Examples, 0, Num_pos),
        length(Examples, Total),
        subtract(Total, Num_pos, Num_neg),
        max(Num_pos, Num_neg, Max),
        divide(Max, Total, Purity).

average_purities(Purities, Purity) :-
        sum(Purities, 0, Sum),
        length(Purities, Number),
        divide(Sum, Number, Purity).
```

```
count_positives([], Num, Num).
count_positives([Example|Examples], Num_in, Num_out) :-
        Example = [positive, _],
        add(1, Num_in, Num_temp),
        count_positives(Examples, Num_temp, Num_out).
count_positives([Example|Examples], Num_in, Num_out) :-
        Example \= [positive, _],
        count_positives(Examples, Num_in, Num_out).

/* Return all examples that have a particular attribute value */
attval_subset(Name, Val, [], []).
attval_subset(Name, Val, [Example|Examples], [Example|Subset]) :-
        has_attr_val(Example, Name, Val),
        attval_subset(Name, Val, Examples, Subset).
attval_subset(Name, Val, [Example|Examples], Subset) :-
        has_not_attr_val(Example, Name, Val),
        attval_subset(Name, Val, Examples, Subset).

has_attr_val(Example, Name, Value) :-
        Example = [_, Attr_list],
        member([Name, Value], Attr_list).

has_not_attr_val(Example, Name, Value) :-
        Example = [_, Attr_list],
        not_member([Name, Value], Attr_list).

/* delete an element from a list */
fdt_delete(Elt, [Elt|Rest], Rest).
fdt_delete(Elt, [Elt2|Elts], [Elt2|Rest]) :-
        Elt \= Elt2,
        fdt_delete(Elt, Elts, Rest).

/* all examples in a set are in the same class */
pure([]).
pure([_]).
pure([Example1, Example2 | Examples]) :-
        Example1 = [Class,_],
        Example2 = [Class,_],
        pure([Example2|Examples]).
```

```
/* all examples in a set are not in the same class */
impure([Example1 | Examples]) :-
        Example1 = [Class1,_],
        member(Example2, Examples),
        Example2 = [Class2,_],
        Class1 \= Class2,
        !.

/* miscellaneous utility predicates */
sum([], Sum, Sum).
sum([Num|Nums], Sum_in, Sum_out) :-
        add(Num, Sum_in, Sum_temp),
        sum(Nums, Sum_temp, Sum_out).

add(A, B, C) :- nonvar(A), nonvar(B), C is A + B.
subtract(A, B, C) :- nonvar(A), nonvar(B), C is A - B.
divide(A, B, C) :- nonvar(A), nonvar(B), B \== 0, C is A / B.
max(A, B, A) :- nonvar(A), nonvar(B), A >= B, !.
max(A, B, B) :- nonvar(A), nonvar(B), A < B.
not_member(X, Y) :- \+ member(X, Y).
less(X,Y) :- X < Y.
less_or_equal(X,Y) :- X =< Y.
```

G.2 SAMPLE RUN

```
Forte version 2.15 -- Test parameters

Data set, initial theory, and domain knowledge:  ../Data/dtree1,
                                        ../Data/dtree1-ca4, ../Domain/dtree1

The data set contains 26 instances.
Language bias is:  [depth_limit(15),use_relations,use_theory,use_built_in,recursive,
                                              relation_tuning(non_relational)]

Output detail set to 7, and initial random seed is random

Test series of 1 independent runs
Training set is 26 instances; test set is 0 instances

Initial Theory

choose_attribute(A,B,C,D,D).
choose_attribute([A|B],C,D,E,F):-
        purity(A,C,G),less_or_equal(G,D),choose_attribute(B,C,D,E,F).
choose_attribute([A|B],C,D,E,F):-purity(A,C,G),less(D,G),choose_attribute(B,C,G,A,F).
```

```
induce_tree(A,[],B,[[C,D]|E],C):-impure([[C,D]|E]).
induce_tree(A,[],B,[],[]).
induce_tree(A,[B,C|D],E,F,[[B,G]|H]):-
        attval_subset(A,B,F,I),induce_tree(J,[],E,I,G),induce_tree(A,[C|D],E,F,H).
induce_tree(A,[B],C,D,[[B,E]]):-attval_subset(A,B,D,F),induce_tree(G,[],C,F,E).
induce_tree(A,[],B,[C|D],[E,F]):-
        impure([C|D]),choose_attribute(B,[C|D],0,G,[E,H]),
        fdt_delete([E,H],B,I),induce_tree(E,H,I,[C|D],F).

induce_tree(A,B,C):-induce_tree(D,[],A,B,C).

--------- Beginning test run with seed:  random(9690,2928,10754,425005073) ---------

Initial train/test accuracy:  0.346154, 0

Selected revision by delete_antecedent scores (12, 1) by replacing the old
induce_tree(A,H,B,C,D):-H=[],C=[E|F],impure(C),E=[D,G].
induce_tree(A,C,B,D,E):-C=[],D=[],E=[].
induce_tree(A,L,D,E,M):-
        L=[B|C],M=[[B,F]|G],C=[H|I],attval_subset(A,B,E,J),
        induce_tree(K,[],D,J,F),induce_tree(A,C,D,E,G).
induce_tree(A,H,C,D,I):-H=[B],I=[[B,E]],attval_subset(A,B,D,F),induce_tree(G,[],C,F,E).
induce_tree(A,L,B,C,M):-
        L=[],M=[D,E],C=[F|G],impure(C),choose_attribute(B,C,0,H,I),
        fdt_delete(I,B,J),I=[D,K],induce_tree(D,K,J,C,E).
with the new
induce_tree(A,H,B,C,D):-H=[],C=[E|F],E=[D,G].
induce_tree(A,C,B,D,E):-C=[],D=[],E=[].
induce_tree(A,L,D,E,M):-
        L=[B|C],M=[[B,F]|G],C=[H|I],attval_subset(A,B,E,J),
        induce_tree(K,[],D,J,F),induce_tree(A,C,D,E,G).
induce_tree(A,H,C,D,I):-H=[B],I=[[B,E]],attval_subset(A,B,D,F),induce_tree(G,[],C,F,E).
induce_tree(A,L,B,C,M):-
        L=[],M=[D,E],C=[F|G],impure(C),choose_attribute(B,C,0,H,I),
        fdt_delete(I,B,J),I=[D,K],induce_tree(D,K,J,C,E).
end revision

Selected revision by add_antecedent scores (4, -1) by replacing the old
choose_attribute(A,B,C,D,D).
choose_attribute(H,C,D,E,F):-
        H=[A|B],purity(A,C,G),less_or_equal(G,D),choose_attribute(B,C,D,E,F).
choose_attribute(H,C,D,E,F):-H=[A|B],purity(A,C,G),less(D,G),choose_attribute(B,C,G,A,F).
with the new
choose_attribute(A,B,C,D,D):-A=[].
choose_attribute(H,C,D,E,F):-
        H=[A|B],purity(A,C,G),less_or_equal(G,D),choose_attribute(B,C,D,E,F).
choose_attribute(H,C,D,E,F):-H=[A|B],purity(A,C,G),less(D,G),choose_attribute(B,C,G,A,F).
end revision
```

```
Selected revision by add_antecedent scores (1, -1) by replacing the old
induce_tree(A,F,B,G,C):-F=[],G=[[C,D]|E].
induce_tree(A,C,B,D,E):-C=[],D=[],E=[].
induce_tree(A,K,E,F,L):-
        K=[B,C|D],L=[[B,G]|H],attval_subset(A,B,F,I),
        induce_tree(J,[],E,I,G),induce_tree(A,[C|D],E,F,H).
induce_tree(A,H,C,D,I):-H=[B],I=[[B,E]],attval_subset(A,B,D,F),induce_tree(G,[],C,F,E).
induce_tree(A,J,B,K,L):-
        J=[],K=[C|D],L=[E,F],impure([C|D]),choose_attribute(B,[C|D],0,G,[E,H]),
        fdt_delete([E,H],B,I),induce_tree(E,H,I,[C|D],F).
with the new
induce_tree(A,F,B,G,C):-F=[],G=[[C,D]|E],pure(E).
induce_tree(A,C,B,D,E):-C=[],D=[],E=[].
induce_tree(A,K,E,F,L):-
        K=[B,C|D],L=[[B,G]|H],attval_subset(A,B,F,I),
        induce_tree(J,[],E,I,G),induce_tree(A,[C|D],E,F,H).
induce_tree(A,H,C,D,I):-H=[B],I=[[B,E]],attval_subset(A,B,D,F),induce_tree(G,[],C,F,E).
induce_tree(A,J,B,K,L):-
        J=[],K=[C|D],L=[E,F],impure([C|D]),choose_attribute(B,[C|D],0,G,[E,H]),
        fdt_delete([E,H],B,I),induce_tree(E,H,I,[C|D],F).
end revision

Revised Theory

choose_attribute([],A,B,C,C).
choose_attribute([A|B],C,D,E,F):-purity(A,C,G),less_or_equal(G,D),choose_attribute(B,C,D,E,F).
choose_attribute([A|B],C,D,E,F):-purity(A,C,G),less(D,G),choose_attribute(B,C,G,A,F).

induce_tree(A,[],B,[[C,D]|E],C):-pure(E).
induce_tree(A,[],B,[],[]).
induce_tree(A,[B,C|D],E,F,[[B,G]|H]):-
        attval_subset(A,B,F,I),induce_tree(J,[],E,I,G),induce_tree(A,[C|D],E,F,H).
induce_tree(A,[B],C,D,[[B,E]]):-attval_subset(A,B,D,F),induce_tree(G,[],C,F,E).
induce_tree(A,[],B,[C|D],[E,F]):-
        impure([C|D]),choose_attribute(B,[C|D],0,G,[E,H]),
        fdt_delete([E,H],B,I),induce_tree(E,H,I,[C|D],F).

induce_tree(A,B,C):-induce_tree(D,[],A,B,C).

                Initial   Initial   Initial               Final     Final     Final
Training         Theory  Training  Test Set  Training    Theory  Training  Test Set
Set Size           Size  Accuracy  Accuracy      Time      Size  Accuracy  Accuracy
--------      ---------  --------  --------  --------  --------  --------  --------
      26             43     34.62      0.00      3011        41    100.00      0.00

--------- End of test run ---------
```

Appendix H
QUALITATIVE MODELLING

This appendix includes a sample of the MISQ domain knowledge contained in the fundamental domain theory, as well as behavioral inputs and FORTE outputs for two cascaded tanks (without netflows) and the RCS.

H.1 DOMAIN KNOWLEDGE

FORTE's domain knowledge for qualitative modelling is a modification of the QSIM constraints coded in MISQ [Richards, Kraan, and Kuipers, 1992]. While the complete code is too lengthy to include here, this section shows the code for one of the simpler constraints: the M+.

```
m_plus(Var1, Var2) :-
  /* only second can be new          */
  get_qualvar(Var1, Behavior1, Units1, New1),
  New1 \== new,
  /* partial var must have qdirs     */
  ( New1 == partial  -> has_qdirs(Behavior1) ; true ),
  get_qualvar(Var2, Behavior2, Units2, New2),
  /* prevent isomorphic duplicates */
  var_order(Var1, Var2),
  /* units checking                  */
  rv:constraint_units(m_plus(Units1, Units2)),
  /* QSIM constraint checking        */
  misq_m_plus(Behavior1, Behavior2),
  /* Is our instantiation valid?    */
  ( New1 \== old  ->  rv:valid_data(Var1) ; true),
  ( New2 \== old  ->  rv:valid_data(Var2) ; true),
  /* Do corr. val's. make sense?    */
  check_corr(m_plus, Behavior1, Behavior2, []).

/* Validity of M+ constraint:  same Qdirs */

misq_m_plus([], []).
misq_m_plus([[_, Qdir, _] | Qvals1], [[_, Qdir, _] | Qvals2]) :-
     Qdir \== ign,
     misq_m_plus(Qvals1, Qvals2).
```

H.2 CASCADED TANKS

```
/* behavior 1 */
example( [ model(amount_a, amount_b, inflow_a, outflow_a, outflow_b) ],
       [ ],
       [ qualvar([ [amount_a,  [0, inf], [[mass],[]],
                          [[0,inc,0], [[0,inf],inc,plus], [a3,std,plus], [a3,std,plus],
                           [a3,std,plus]]],
              [amount_b,  [0, inf], [[mass],[]],
                          [[0,std,0], [[0,inf],inc,plus], [[0,inf],inc,plus],
                           [[0,inf],inc,plus], [b4,std,plus]]],
              [inflow_a,  [0, if, inf], [[mass],[time]],
                          [[if,std,plus], [if,std,plus], [if,std,plus], [if,std,plus],
                           [if,std,plus]]],
              [outflow_a, [0, inf], [[mass],[time]],
                          [[0,inc,0], [[0,inf],inc,plus], [of_a3,std,plus],
                           [of_a3,std,plus], [of_a3,std,plus]]],
              [outflow_b, [0, inf], [[mass],[time]],
                          [[0,std,0], [[0,inf],inc,plus], [[0,inf],inc,plus],
                           [[0,inf],inc,plus], [of_b4,std,plus]]]
            ])
       ],
  facts([ ])
).

/* behavior 2 */
example( [ model(amount_a, amount_b, inflow_a, outflow_a, outflow_b) ],
       [ ],
       [ qualvar([ [amount_a,  [0, inf], [[mass],[]],
                          [[0,inc,0], [[0,inf],inc,plus], [a2,std,plus]]],
              [amount_b,  [0, inf], [[mass],[]],
                          [[0,std,0], [[0,inf],inc,plus], [b2,std,plus]]],
              [inflow_a,  [0, if, inf], [[mass],[time]],
                          [[if,std,plus], [if,std,plus], [if,std,plus]]],
              [outflow_a, [0, inf], [[mass],[time]],
                          [[0,inc,0], [[0,inf],inc,plus], [of_a2,std,plus]]],
              [outflow_b, [0, inf], [[mass],[time]],
                          [[0,std,0], [[0,inf],inc,plus], [of_b2,std,plus]]]
            ])
       ],
  facts([ ])
).
```

Induced model

```
model(A,B,C,D,E) :-
        m_plus(B,E),
        m_plus(A,D),
        constant(C),
        derivative(B,F),
        add(E,F,D),
        add(D,G,C),
        m_minus(A,G),
        derivative(A,G),
        m_minus(D,G).
```

H.3 REACTION CONTROL SYSTEM

```
/* behavior 1 */
example( [ model(amt_he, p_he, d_amt_he, he_ull_flow, amt_ull, p_ull, amt_fuel,
               vol_fuel, vol_ull, vol_total, den_fuel, d_amt_fuel,
               p_diff_fuel, amt_man, d_amt_man, p_man) ],
        [ ],
 [      qv([ [ amt_he,       [0, amt_he_min, amt_he_preg, amt_he_sreg, amt_he_init],
                          [[mass_he],[]],
                          [ [ amt_he_sreg,            dec, plus],
                            [ [amt_he_preg, amt_he_sreg], dec, plus],
                            [ a9,                   dec, plus] ]],
           [ p_he,         [0, p_he_min, p_he_preg, p_he_sreg, p_he_init],
                          [[mass_he],[distance_he_tank,time,time]],
                          [ [ p_he_sreg,              dec, plus],
                            [ [p_he_preg, p_he_sreg],    dec, plus],
                            [ p6,                   dec, plus] ]],
           [ d_amt_he,     [minf, 0, inf],
                          [[mass_he],[time]],
                          [ [ [minf, 0],              ign, minus],
                            [ [minf, 0],              ign, minus],
                            [ [minf, 0],              ign, minus] ]],
           [ he_ull_flow,  [0, inf],
                          [[mass_he],[time]],
                          [ [ [0, inf],               ign, plus],
                            [ [0, inf],               ign, plus],
                            [ [0, inf],               ign, plus] ]],
           [ amt_ull,      [0, amt_ull_init, amt_ull_nom, amt_ull_max],
                          [[mass_he],[]],
                          [ [ amt_ull_nom,               inc, plus],
                            [ [amt_ull_nom, amt_ull_max],  inc, plus],
                            [ a-10,                      inc, plus] ]],
[ p_ull,        [0, p_ull_preg],
               [[mass_he],[distance_he,time,time]],
               [ [ p_ull_preg,            std, plus],
                 [ p_ull_preg,            std, plus],
                 [ p_ull_preg,            std, plus] ]],
[ amt_fuel,     [0, amt_f_nom, amt_f_init],
               [[mass_fuel],[]],
               [ [ amt_f_nom,              dec, plus],
                 [ [0, amt_f_nom],          dec, plus],
                 [ 0,                   dec, 0] ]],
[ vol_fuel,     [0, vf_nom, vf_init, vol_total],
               [[distance_tank,distance_tank,distance_tank],[]],
               [ [ vf_nom,                dec, plus],
                 [ [0, vf_nom],             dec, plus],
                 [ 0,                   dec, 0] ]],
[ vol_ull,      [0, vu_init, vu_nom, vol_total],
               [[distance_tank,distance_tank,distance_tank],[]],
               [ [ vu_nom,                inc, plus],
       [ [vu_nom, vol_total],        inc, plus],
                 [ vol_total,             inc, plus] ]],
[ vol_total,    [0, vol_total, inf],
               [[distance_tank,distance_tank,distance_tank],[]],
               [ [ vol_total,             std, plus],
                 [ vol_total,             std, plus],
                 [ vol_total,             std, plus] ]],
[ den_fuel,     [0, dfpreg, dfmax],
               [[mass_fuel],[distance_tank,distance_tank,distance_tank]],
               [ [ dfpreg,                std, plus],
                 [ dfpreg,                std, plus],
                 [ dfpreg,                std, plus] ]],
[ d_amt_fuel,   [minf, 0, inf],
               [[mass_fuel],[time]],
               [ [ [minf, 0],             ign, minus],
                 [ [minf, 0],             ign, minus],
                 [ [minf, 0],             ign, minus] ]],
[ p_diff_fuel,  [minf, 0, inf],
               [[mass_he],[distance_he,time,time]],
               [ [ [0, inf],              dec, plus],
                 [ [0, inf],              dec, plus],
                 [ [0, inf],              dec, plus] ]],
[ amt_man,      [0, a0, amt_preg],
               [[mass_fuel],[]],
               [ [ a0,                    inc, plus],
                 [ [a0, amt_preg],            inc, plus],
                 [ a11,                   inc, plus] ]],
      [ d_amt_man,     [minf, 0, inf],
                       [[mass_fuel], [time]],
                       [ [ [0, inf],             ign, plus],
                         [ [0, inf],             ign, plus],
                         [ [0, inf],             ign, plus] ]],
      [ p_man,           [0, p1, p_man_preg],
                       [[mass_he],[distance_he,time,time]],
                       [ [ p1,                   inc, plus],
                         [ [p1, p_man_preg],           inc, plus],
                         [ p8,                   inc, plus] ]]
      ])
    ],
  facts([ ])
).
```

<u>Induced model</u>

```
model(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P) :-
        m_plus(P,I), m_plus(N,P), m_plus(N,I), m_plus(N,E), m_plus(M,H),
        m_plus(M,B), m_plus(K,J), m_plus(K,F), m_plus(G,M), m_plus(G,H),
        m_plus(G,B), m_plus(G,A), m_plus(F,J), m_plus(E,P), m_plus(E,I),
        m_plus(B,H), m_plus(A,M), m_plus(A,H), m_plus(A,B),
        m_minus(P,H), m_minus(N,M), m_minus(N,H), m_minus(N,B),
        m_minus(M,P), m_minus(M,I), m_minus(K,J), m_minus(K,F),
        m_minus(H,I), m_minus(G,P), m_minus(G,N), m_minus(G,I),
        m_minus(G,E), m_minus(F,J), m_minus(E,M), m_minus(E,H),
        m_minus(E,B), m_minus(B,P), m_minus(B,I), m_minus(A,P),
        m_minus(A,N), m_minus(A,I), m_minus(A,E),
        derivative(N,O), derivative(G,L), derivative(E,D), derivative(A,C),
        mult(K,I,N), mult(K,H,G),
        minus(L,O), minus(C,D),
        add(M,P,F), add(H,I,J),
        constant(K), constant(J), constant(F).
```

BIBLIOGRAPHY

R. Bhaskar and A. Nigam, "Qualitative Physics Using Dimensional Analysis," *Artificial Intelligence*, 45:73-111, 1990.

M. Bain, "Experiments in non-monotonic learning," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 380-384, 1991.

I. Bratko, S. Muggleton, and A. Varšek, "Learning Qualitative Models of Dynamic Systems," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 385-388, 1991.

A. Bundy, A. Smaill, and G. Wiggins, "The Synthesis of Logic Programs from Inductive Proofs," in *Computational Logic, Symposium Proceedings*, pp. 135-149, Springer-Verlag, 1990.

W. Buntine, "Myths and Legends in Learning Classification Rules," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-1990), pp. 736-742, 1990.

W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, second edition, Springer-Verlag, 1984.

W. W. Cohen, "Compiling Prior Knowledge into an Explicit Bias," *Proceedings of the Ninth International Conference on Machine Learning*, 1992.

W. W. Cohen, "Grammatically Biased Learning: Learning Horn Theories Using an Explicit Antecedent Description Language," Technical Memorandum 11262-910708-16TM, AT&T Bell Laboratories, 1991.

E. Coiera, "Generating Qualitative Models from Example Behaviors," Technical Report DCS 8901, Department of Computer Science, University of New South Wales, May 1989.

S. Craw and D. Sleeman, "The Flexibility of Speculative Refinement," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 28-32, 1991.

J. Crawford, A. Farquhar, and B. Kuipers, "QPC: A Compiler from Physical Models into Qualitative Differential Equations," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), pp. 365-372, 1990.

D. DeCoste, "Dynamic Across-Time Measurement Interpretation," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), pp. 373-379, 1990.

J. deKleer, "An assumption-based truth maintenance system," "Extending the ATMS," "Problem solving with the ATMS," *Artificial Intelligence*, 28(2):127-224, 1986.

J. deKleer and J. Brown, "A Qualitative Physics Based on Confluences," *Artificial Intelligence*, 24:7-83, 1984.

B. Falkenhainer and K. Forbus, "Compositional Modeling: Finding the Right Model for the Job," unpublished draft, 1990.

E. A. Feigenbaum, "The Art of Artificial Intelligence: I. Themes and Case Studies of Knowledge Engineering," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (IJCAI-77), pp. 1014-1029, 1977.

P. Flener and Y. Deville, "Toward stepwise, schema-guided synthesis of logic programs," *Proceedings of the Workshop on Logic Program Synthesis and Transformation* (LOPSTR-91), pp. 46-64, 1991.

K. Forbus, "Qualitative Process Theory," *Artificial Intelligence*, 24:85-168, 1984.

K. Forbus, "The Qualitative Process Engine," Technical Report, Department of Computer Science, University of Illinois, 1986.

K. Forbus and B. Falkenhainer, "Setting Up Large-Scale Qualitative Models," *Proceedings of the Seventh National Conference on Artificial Intelligence* (AAAI-88), pp. 301-306, 1988.

K. Forbus, B. Falkenhainer, "Self-Explanatory Simulations: An integration of qualitative and quantitative knowledge," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), pp. 380-387, 1990.

A. Ginsberg, "Theory Reduction, Theory Revision, and Retranslation," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-1990), pp. 777-782, 1990.

D. Haussler, "Quantifying Inductive Bias: AI Learning Algorithms and Valiant's Learning Framework," *Artificial Intelligence*, 36:177-221, 1988.

J. Hellerstein, "Obtaining Quantitative Predictions from Monotone Relationships," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), pp. 388-394, 1990.

G. E. Hinton, "Learning Distributed Representations of Concepts," *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 1986.

H. Kay, "A Qualitative Model of the Space Shuttle Reaction Control System," Technical Report, Department of Computer Sciences, University of Texas at Austin, 1992.

B. Kijsirikul, M. Numao, and M. Shimura, "Efficient Learning of Logic Programs with Non-determinate, Non-descriminating Literals," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 417-421, 1991.

I. Kraan, D. Basin, and A. Bundy, "Logic Program Synthesis via Proof Planning," *Proceedings of the Workshop on Logic Program Synthesis and Transformation* (LOPSTR-92), 1992.

I. C. Kraan, B. L. Richards, and B. J. Kuipers, "Automatic Abduction of Qualitative Models," *Proceedings of the Fifth International Workshop on Qualitative Reasoning about Physical Systems*, pp. 295-301, 1991.

B. Kuipers, "Qualitative Reasoning: Modelling and Simulation with Incomplete Knowledge," *Automatica*, 25:571-585, 1989.

B. Kuipers, *Qualitative Reasoning: Modeling and Simulation with Incomplete Knowledge*, unpublished draft, 1990.

B. Kuipers and D. Berleant, "Using Incomplete Quantitative Knowledge in Qualitative Reasoning," *Proceeding of the Seventh National Conference on Artificial Intelligence* (AAAI-88), pp. 324-329, 1988.

B. Kuipers, "Qualitative Simulation," *Artificial Intelligence*, 29:289-338, 1986.

B. Kuipers, "Causal Reasoning in Medicine: Analysis of a Protocol," *Cognitive Science*, 8:363-385, 1984.

P. Langley, "Finding Common Paths as a Learning Mechanism," Carnegie-Mellon University CIP Working Paper No. 419, 1980.

P. Langley, "Rediscovering Physics with BACON.3," *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (IJCAI-77), pp. 505-507, 1977.

P. Langley, "Finding Common Paths as a Learning Mechanism," CIP Working Paper #419, Carnegie-Mellon University, 1980.

J. W. Lloyd, *Foundations of Logic Programming*, second, extended edition, Springer-Verlag, 1987.

J. W. Lloyd and R. W. Torpor, "Making Prolog more Expressive," *Journal of Logic Programming*, Vol.1, No. 3, pp. 225-240, 1984.

J. J. Mahoney and R. J. Mooney, "Combining Symbolic and Neural Learning to Revise Probabilistic Theories," *Working Notes of the Machine Learning Workshop on Integrated Learning in Real-World Domains*, Aberdeen, Scotland, July 1992.

T. M. Mitchell, "Generalization as Search," *Artificial Intelligence*, 18:203-266, 1982.

R. J. Mooney and B. L. Richards, "Automated Debugging of Logic Programs via Theory Revision," *Proceedings of the Second International Workshop on Inductive Logic Programming*, Tokyo, Japan, June 1992.

S. Muggleton, "Duce, an Oracle based approach to constructive induction," *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (IJCAI-87), pp. 287-292, 1987.

S. Muggleton, "Inductive Logic Programming," *Proceedings of the Workshop on Algorithmic Learning Theory* (ALT-90), pp. 42-62, 1990.

S. Muggleton and W. Buntine, "Machine Invention of First-order Predicates by Inverting Resolution," *Proceedings of the Fifth International Conference on Machine Learning*, pp. 339-352, 1988

S. Muggleton and C. Feng, "Efficient induction of logic programs," *Proceedings of the First Conference on Algorithmic Learning Theory*, 1990.

W. R. Murray, "Automatic Program Debugging for Intelligent Tutoring Systems," Technical Report AI TR86-27, Univerisy of Texas at Austin, 1986.

H. T. Ng and R. J. Mooney, "On the Role of Coherence in Abductive Explanation," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), pp. 337-342, 1990.

D. Ourston and R. J. Mooney, "Changing the Rules: A Comprehensive Approach to Theory Refinement," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), pp. 815-820, 1990.

M. J. Pazzani and C. A. Brunk, "Detecting and Correcting Errors in Rule-Based Expert Systems: An Integration of Empirical and Explanation-based Learning," *Knowledge Acquisition*, 3:157-173, 1991.

M. J. Pazzani, C. A. Brunk, and G. Silverstein, "A knowledge-intensive approach to relational concept learning," *Proceedings of the Eighth International Workshop on Machine Learning*, 1991.

M. J. Pazzani and D. Kibler, "The Utility of Prior Knowledge in Inductive Learning," *Machine Learning*, 9:57-94, 1992.

M. R. Quillian, "Semantic Memory," *Semantic Information Processing*, MIT Press, pp. 227-270, 1968.

J. R. Quinlan, "Determinate Literals in Inductive Logic Programming," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 442-446, 1991.

J. R. Quinlan, "Induction of Decision Trees," *Machine Learning*, 1:81-106, 1986.

J. R. Quinlan, "Learning Logical Definitions from Relations," *Machine Learning*, 5:239-266, 1990.

B. L. Richards, I. Kraan, and B. J. Kuipers, "Automatic Abduction of Qualitative Models," *Proceedings of the Tenth National Conference on Artificial Intelligence* (AAAI-92), 1992.

B. L. Richards and R. J. Mooney, "First-Order Theory Revision," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 447-451, 1991.

B. L. Richards and R. J. Mooney, "Learning Relations by Pathfinding," *Proceedings of the Tenth National Conference on Artificial Intelligence* (AAAI-92), 1992.

E. Shapiro, *Algorithmic Program Debugging*, MIT Press, 1983.

G. Silverstein and M. J. Pazzani, "Relational clichés: Constraining constructive induction during relational learning," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 203-207, 1991.

R. F. Simmons, Y. H. Yu, "The Acquisition and Application of Context Sensitive Grammar for English," *Proceedings of the 29th Annual Meeting of the Association for Computational Linguistics*, 1991.

H. Tamaki and T. Sato, "A Transformation System for Logic Programs that Preserves Equivalence," ICOT Technical Report TR-018, 1984.

G. G. Towell, J. W. Shavlik, and M. W. Craven, "Interpretation of Artificial Neural Networks: Mapping Knowledge-Based Neural Networks into Rules," *Proceedings of the Eighth International Workshop on Machine Learning*, 1991.

G. G. Towell, J. W. Shavlik, and M. O. Noordewier, "Refinement of Approximate Domain Theories by Knowledge-Based Neural Networks," *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-1990), pp. 861-866, 1990.

S. A. Vere, "Induction of Relational Productions in the Presence of Background Information," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (IJCAI 1977), pp. 349-355, 1977.

D. A. Waterman, *A Guide to Expert Systems*, Addison Wesley, 1986.

R. Wirth, "Completing Logic Programs by Inverse Resolution," *Proceedings of the Fourth European Working Session on Learning* (EWSL-89), pp. 239-250, 1989.

R. Wirth and P. O'Rorke, "Constraints on Predicate Invention," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 457-461, 1991.

J. Wogulis, "Revising Relational Domain Theories," *Proceedings of the Eighth International Workshop on Machine Learning*, pp. 462-466, 1991.

J. Zelle and R. J. Mooney, "Speeding up Logic Programs by Combining EBG and FOIL," *Working Notes of the Machine Learning Workshop on Knowledge Compilation and Speedup Learning*, Aberdeen, Scotland, July 1992.