University of Kentucky
UKnowledge

University of Kentucky Master's Theses | Graduate School

2003

REVISING HORN FORMULAS

Jignesh Umesh Doshi
University of Kentucky, jigud@hotmail.com

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Doshi, Jignesh Umesh, "REVISING HORN FORMULAS" (2003). *University of Kentucky Master's Theses*. 222.
https://uknowledge.uky.edu/gradschool_theses/222

This Thesis is brought to you for free and open access by the Graduate School at UKnowledge. It has been accepted for inclusion in University of Kentucky Master's Theses by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

ABSTRACT OF THESIS

REVISING HORN FORMULAS

Boolean formulas can be used to model real-world facts. In some situation we may have a Boolean formula that closely approximates a real-world fact, but we need to fine-tune it so that it models the real-world fact exactly. This is a problem of theory revision where the theory is in the form of a Boolean formula. An algorithm is presented for revising a class of Boolean formulas that are expressible as conjunctions of Horn clauses. Each of the clauses in the formulas considered here has a unique unnegated variable that does not appear in any other clauses, and is not 'F'. The revision algorithm uses equivalence and membership queries to revise a given formula into a formula that is equivalent to an unknown target formula having the same set of unnegated variables. The amount of time required by the algorithm to perform this revision is logarithmic in the number of variables, and polynomial in the number of clauses in the unknown formula. An early version of this work was presented at the 2003 Midwest Artificial Intelligence and Cognitive Science Conference [4].

KEYWORDS: Propositional Horn sentences, Horn formulas, theory revision, equivalence queries, membership queries.

Jignesh Umesh Doshi

August 6, 2003.

REVISING HORN FORMULAS

By

Jignesh Umesh Doshi

Dr. Judy Goldsmith
(Director Of Thesis)

Dr. Grzegorz W. Wasilkowski
(Director Of Graduate Studies)

August 6, 2003.

RULES FOR THE USE OF THESES

Unpublished theses submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgments.

Extensive copying or publication of the thesis in whole or in part requires also the consent of the Dean of The Graduate School of the University of Kentucky.

THESIS

Jignesh Umesh Doshi

The Graduate School
University of Kentucky
2003

REVISING HORN FORMULAS

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science
in the College of Engineering
at the University of Kentucky

By

Jignesh Umesh Doshi

Lexington, Kentucky

Director: Dr. Judy Goldsmith, Professor of Computer Science

Lexington, Kentucky

2003

Copyright ©Jignesh U. Doshi 2003.

Acknowledgments

First, I want to thank my Thesis Chair, Dr. Judy Goldsmith, for her continuous guidance and support. This includes not only the period of the creation of this Thesis, but also throughout my master's studies and research at the University of Kentucky. Her faith in me and the freedom she provided have made this work possible.

I would also like to thank Dr. Robert Sloan at the University of Illinois, Chicago, and the Midwest Artificial Intelligence and Cognitive Science Conference (MAICS-2003) referees, for reading through and pointing out improvements in my paper, which provided the theoretical base for this work. I thank György Turán for help in implementing the equivalence query oracle.

Assistance from my parents and brothers was extremely helpful during my stay here. All my professors, Dr. Raphael Finkel, Dr. Ken Calvert, Dr. Andrew Klapper and others, have made my studies at the University of Kentucky an enriching experience. I would like to thank everyone at the CS Department with an emphasis on Dr. Judy Goldsmith for her trust in me, and her continuous help from my first day at the University of Kentucky.

Table of Contents

List of Figures

List of Files

1. JDThesis.pdf 499 KB

Chapter 1

Introduction

Suppose we have a good estimate of what an unknown target theory may be, but don't know it exactly, and would like to revise this theory to always be correct. What we need is a revision algorithm, and not a learning algorithm. We can always use what we already know (a known initial theory) while revising our hypothesis so that it becomes equivalent to an unknown target theory. I present an algorithm for revising a subclass of propositional Horn formulas (a subclass of Boolean formulas), in which revisions consist of deletion or addition of Horn clause body variables. My algorithm is polynomial in the number of clauses in the formula and logarithmic in the number of variables in the universe. My work extends the positive results for revising Horn formulas in [6] in one significant direction. My construction allows addition of new body variables in the revision process. The algorithm presented in this thesis, however, deals with a subclass of Horn formulas that I define as *Unique Explanations* (definition on page 9).

The problem solved here belongs to the field of computational learning theory. In Section 1.1, I give an overview of the field of computational learning theory. A very brief text on the entities involved in computational learning theory is given there. I explain why this field was developed and how it is related to the work in this thesis. In Section 1.2, I explain learning with queries, a subfield of computational learning theory, along with the entities involved in the learning process and the types of queries posed. In Section 1.3, I explain the concept of theory revision with queries. Section 1.4 mentions the problem I solved and the revision model used in this thesis. I conclude this chapter with an outline of the rest of the thesis.

1.1 Computational Learning Theory

The following information was collected from the book "Machine Learning" [12].

Computational learning theory is the design and analysis of algorithms for making predictions on the basis of past experiences. In computational learning theory, one defines a formal mathematical model for learning. A formal model is chosen based on the following criteria: What is being learned? What is the environment like; that is, is there a teacher? and What is known till now? Computational learning theory involves answering questions like: How many examples are necessary or sufficient to assure successful learning? or How is the number of examples required affected in settings where: a learner poses queries to a teacher, a teacher chooses examples or a teacher labels randomly generated instances.

Generally the attributes of a computational learning problem are the number of training examples required to learn, the complexity of the hypothesis space, and the manner in which training examples are presented. If c is a theory and x is an example, then training examples are usually presented in any one of the following ways:

- A learner proposes instances as queries to a teacher, that is, a learner proposes an instance x and a teacher provides $c(x)$(the value of c for an instance x).

- A teacher (that knows c) provides training examples, that is, a teacher provides a sequence of examples of the form (x, $c(x)$).

- Some random process (for example, nature) proposes the instances, that is, an instance x is generated randomly and a teacher provides $c(x)$.

Next, I mention the values of these attributes for the problem solved here. In the work presented in this thesis, the number of training examples required to revise a theory is determined by the number of queries posed by a learner to a teacher. The hypothesis space in my work is a subclass of Boolean formulas. In the problem solved here, the teacher knows c and provides training examples to a learner. Thus, the work in this thesis is about computation learning in a particular setting. The solution presented in this thesis works for a subclass of computational learning problems in which queries are used for learning.

1.2 Learning with Queries

In the Learning-with-Queries model, a learner tries to identify a target theory through queries to a teacher. This model is also known as Angluin's model of learning with queries [2]. Learning with queries involves two entities: queries and teachers. In this type of learning, a learner poses queries to the teachers that always answer them correctly. Two types of teachers for learning a subclass of Boolean formulas, *membership oracles* and *equivalence oracles*, were first introduced by Angluin, et al. in [2].

In a query to an *equivalence oracle*, the learning algorithm proposes a hypothesis h, and the answer depends on whether the hypothesis is equivalent to the target theory. If so, the answer is 'Yes', and the learning algorithm has succeeded in its goal of exact identification of the target theory. Otherwise, the answer is a counterexample: some instance x such that x is true for the hypothesis and false for the target theory or vice versa.

In a query to a *membership oracle*, the learning algorithm gives an instance x, and the answer is either false or true depending on whether x is false or true for the target. Membership queries work on instances, whereas equivalence queries work on formulas.

Instead of learning, I use these types of teachers to revise a subclass of Boolean formulas. Thus, more specifically, the problem I solve in this thesis belongs to a field called "Theory Revision with Queries".

1.3 Theory Revision with Queries

Many times one has a rough explanation that requires alteration. For example, I ask a field expert and an information engineer to develop an expert system for predicting my college roommate's cooking habits. I observe that my roommate cooks only when he is hungry, there is no fight between us, there are no leftovers and it's his turn to cook. Based on this explanation, the following "pretty close" initial theory is developed:

HeIsHungry AND NoFight AND HisTurn AND NoLeftOvers → WillCook.

Then, though I use this initial theory as a general guide, I happen to observe that my roommate didn't cook even though he was hungry, there was no fight before his turn, and there were no leftovers. He had been invited to his friend's place. I must revise or edit my initial theory, perhaps to:

HeIsHungry AND NoFight AND HisTurn AND NoLeftOvers AND NotInvited → WillCook.

This is a problem known in machine learning as theory revision.

Given a Boolean formula that is approximately correct, how does one make it entirely correct? If a given formula differs from an unknown target formula, what is the complexity of revising this given formula? These are some of the questions that arise in theory revision. The term "theory revision" refers to a wide variety of approaches, as in [5] and [7]. Because of the breadth of the field, I do not begin to survey those approaches, but instead limit my survey to work based on *(a)* theory revision with queries, and *(b)* Horn formulas, which are a subclass of Boolean formulas. In fact, the problem of revising Horn formulas has many interpretations. The different interpretations are surveyed by Goldsmith, et al. in [7].

For determining the revisions, teachers are available to which we pose membership queries of the form "Is this an instance of the unknown target formula?", and equivalence queries of the form "Is this hypothesis equivalent to the target formula?". If the hypothesis is not equivalent to the target, an equivalence query returns an instance that is *true* for the hypothesis and *false* for the target or vice versa. These queries are the same queries posed in Angluin's model [2]. However, here these queries are used for revision, and not for learning. Revision algorithms are permitted time (measured as the number of examples) polynomial in the number of required revisions, but only logarithmic in the total number of variables in the universe. Under these limitations, it will generally be faster to revise an existing explanation rather than to start learning from scratch.

Theory revision is of independent interest to the greater computer science community. Unfortunately, most very large theories based on real–world examples are not exactly correct. My work shows that, at least for certain forms of theories, a few corrections to the theory can be made at a much lower cost in terms of examples (i.e., queries in formal models) than (re)learning the whole theory from scratch.

Learning or revising of Horn formulas can be useful in the field of data mining where collection and refinement of information is widely done. Other applications of revision of Horn formulas are in the fields of machine learning, especially in robotics and computer vision. Horn formulas are a common choice for practical machine learning systems, and almost all systems that learn logic programs. Horn formulas are used to represent set operations and are the basis for logic programming language PROLOG. For more information

on the importance of Horn formulas consult [11].

1.4 The Problem

The problem that I solve in this thesis is a theory revision with queries problem in a special setting. The solution presented here deals with the situation where a learner has a rough estimate of what to learn, and where the learner tries to learn an unknown target theory exactly within a specific time complexity. This solution requires that both the initial theory and the target theory satisfy certain conditions. The theories that satisfy these conditions are defined as *Unique Explanations* (definition on page 9).

There are many models for theory revision. Sloan and Turán [15] formulated the theory revision problem in the model of membership and equivalence queries [1], and I am using that model here as well. That is, there is one teacher MQ that tells whether any given instance is positive or negative according to the correct theory, and a second teacher EQ that responds to a *false* conjecture about the correct theory with a counterexample. An expert would function as a membership oracle in an expert system setting.

1.5 Thesis Outline

Chapter 2 gives some basic definitions from logic and learning theory along with some examples. Section 2.3 defines a Horn clause body and a Horn clause head and mentions observations related to Horn clauses. Section 2.4 formally specifies my assumption under which the algorithm in this thesis works correctly.

Chapter 3 introduces other authors' work that is closely related to my thesis. This chapter explains Angluin, et al.'s work on learning Horn formulas [2] and Goldsmith, et al.'s work on revision of Horn formulas [6]. Some important issues and an example run of their algorithm are presented here.

Chapter 4 specifies a road-map towards designing an Additions-Also algorithm. An overview of my algorithm is also presented here. The notation specified in this chapter is consistently used in the pseudo-code of my algorithm, in the explanation and in all the following sections.

Chapter 5 presents the details of each of my algorithm subroutines along with their pseudo-code. Chapter 6 uses this pseudo–code to prove the correctness of lemmas.

Chapter 6 provides the lemmas and their proofs. These lemmas are used later in Chapter 7 for proving the correctness of my algorithm. The complexity derivation in Chapter 7 determines the maximum number of membership queries and equivalence queries required in the revision process.

Chapter 8 covers all the implementation issues of the application designed. This chapter presents information regarding the application design, the application usage, its inputs and outputs, the data structures used and the validations performed. A class diagram of the application design is also shown here.

Chapter 9 shows an actual run of the revision application for an example. This chapter contains a trace of the revision process that shows all the counterexamples encountered along with the queries asked, the answers obtained, the hypotheses built at each stage and the number of equivalence queries and membership queries asked.

Copyright ©Jignesh U. Doshi 2003.

Chapter 2

Definitions

In this chapter, I give some basic definitions from logic (in Section 2.1) and learning theory (in Section 2.2). Section 2.3 provides Horn clause-related definitions and observations. Only definitions that are required to understand the work in this thesis are provided. This chapter concludes with a formal specification of my assumption under which the algorithm in this thesis works correctly. An example that satisfies this assumption, and some examples that don't satisfy this assumption are also provided.

2.1 Basic definitions

I start with some basic definitions from logic.

Horn clause: An implication or disjunction of literals having at most one unnegated variable. For example:

$$a.b.c \rightarrow d \equiv \sim a \vee \sim b \vee \sim c \vee d$$

where a, b, c and d are Boolean variables, '$\sim$' represents a Boolean NOT, '$\vee$' represents a Boolean OR, and '.' represents a Boolean AND.

Horn formula: A conjunction of Horn clauses. For example:

$$(a.c \rightarrow d) \wedge (b.d \rightarrow e) \wedge (c.e \rightarrow f)$$

where a, b, c, d, e and f are Boolean variables, '.' represents a Boolean AND between clause body variables, and '$\wedge$' represents a Boolean AND between two Horn clauses.

Next, I give a couple of general definitions from computational learning theory, as applied to my work.

2.2 Learning Theory–related definitions

The theories we are interested in are the initial theory, the target theory and the hypothesis theory. Below I define an initial theory and a target theory. A hypothesis theory is defined after the definitions of (positive and negative) examples.

Initial theory: A known Horn formula that needs revision.

Target theory: An unknown Horn formula to be derived from the initial theory.

I always work with a finite universe of n variables, so I identify sets of variables with their characteristic bit strings or with their truth assignments. Thus, an example can be represented as a n-bit string but can be intersected with a set of variables. For example,

$$x_1x_3x_4 \cap \mathbf{1101} = x_1x_3x_4 \cap x_1x_2x_4 = x_1x_3x_4 \cap \mathbf{TTFT} = x_1x_4$$

I use *true*/*false* for the answers returned by queries, and T/F or 1/0 for the values assigned to the variables.

Positive and negative examples: A positive (respectively, a negative) example for Horn formula H is an assignment x such that H evaluates to T (respectively, F) when each variable v in H is replaced by a corresponding bit in x.

Let x be an example; then true(x) (respectively, false(x)) is a set consisting of the constant T (respectively, F) and the variables assigned the value T (respectively, F) by x.

Hypothesis: A proposition constructed from the initial theory and the examples obtained from the queries.

Revision distance: The minimal number of deletions or additions of literals needed to obtain the target theory from an initial theory.

Now we know what an example is. Below I define what a counterexample is. Counterexamples can be either negative or positive.

Negative counterexample: An example that does not satisfy the target theory but satisfies the hypothesis.

Positive counterexample: An example that satisfies the target theory but not the hypothesis.

Below I define a membership query and an equivalence query.

Membership Query (MQ): A query that returns *true* if a given example satisfies the target theory, and *false* if a given example does not satisfies the target theory.

Equivalence Query (EQ): A query that returns *true* if the target and the hypothesis theories are equivalent. Equivalent theories need not have the same structure as long as they have the same truth table. If the two theories are not equivalent, the EQ returns a counterexample.

2.3 Horn Clause–related definitions

My algorithm in this thesis revises a subclass of Horn formulas and extensively uses the following two definitions specific to Horn clauses.

Clause's head: A Horn clause's unnegated variable.

Clause's body: A Horn clause's negated variables.

Consider the following Horn clause,

$$a.b.c \rightarrow d \equiv \sim a \vee \sim b \vee \sim c \vee d$$

where a, b, c and d are Boolean variables, '$\sim$' represents a Boolean NOT, '$\vee$' represents a Boolean OR, and '.' represents a Boolean AND. This clause's body is a set $\{a, b, c\}$, and its head is the variable d.

For an example $x \in \{0,1\}^n$ and clause C, we say x *covers* C if body$(C) \subseteq x$, and x *falsifies* C iff x covers C and head$(C) \notin x$. If x and y both cover clause C, and at least one of x and y falsifies C, then $x \cap y$ falsifies C.

2.4 Assumptions

Finally, I mention my assumption for the work presented in this thesis. The algorithm presented in this thesis works correctly on any theory that is a *Unique Explanation* as defined below.

Unique Explanation: A Horn formula in which all the clause heads are:

- **not** the literal **'F'**,
- **unique**,

- **not in any of the bodies**, and
- **intact**. That is, none of these heads is required to be deleted nor is any new head required to be added to the initial Horn formula during the revision process.

For instance, a unique explanation can be used to represent the cooking habits of my roommate. The algorithm presented in this paper revises unique explanations like the one shown below.

Known initial theory:
(HisTurn $\wedge$ NoLeftOvers $\wedge$ HeIsHungry $\rightarrow$ *WillCook*) $\wedge$
(NoFight $\rightarrow$ *NiceMood*) $\wedge$
(NothingElseToDo $\rightarrow$ *EnthuToCook*)
Unknown target theory:
(HisTurn $\wedge$ NoLeftOvers $\wedge$ NotInvited $\rightarrow$ *WillCook*) $\wedge$
(NoFight $\wedge$ PraiseHim $\rightarrow$ *NiceMood*) $\wedge$
(NothingElseToDo $\rightarrow$ *EnthuToCook*)

Conversely, the following theories are not unique explanations:
Known initial theory:
$(b_{11} \wedge b_{12} \wedge b_{13} \rightarrow h_1) \wedge (b_{21} \wedge b_{22} \rightarrow h_1) \wedge (b_{31} \wedge b_{32} \wedge b_{33} \rightarrow h_3)$
Unknown target theory:
$(b_{11} \wedge b_{12} \wedge b_{13} \rightarrow h_1) \wedge (b_{21} \wedge b_{22} \rightarrow h_2) \wedge (b_{31} \wedge b_{32} \wedge b_{33} \rightarrow h_3)$

The initial theory above is not a unique explanation because there are clauses with a non–unique head h_1 in them. Even though the target theory is a unique explanation, the algorithm presented here requires both the initial theory and the target theory to be unique explanations for a successful revision.

Unknown target theory:
$(b_{11} \wedge b_{12} \wedge b_{13} \rightarrow h_1) \wedge (b_{31} \wedge b_{32} \wedge h_1 \rightarrow h_3)$

The target theory above is not a unique explanation as it has a head that is in the body of another clause. No initial theory can be revised to this type of target theory by the algorithm presented in this thesis.

Unknown target theory:

$(b_{11} \wedge b_{12} \wedge b_{13} \rightarrow h_1) \wedge (b_{21} \wedge b_{22} \rightarrow F)$

Literal F is not allowed as a clause head for either of the initial theory or the target theory. The above target theory is therefore not a unique explanation.

Known initial theory:

$(b_{11} \wedge b_{12} \wedge b_{13} \rightarrow h_1) \wedge (b_{21} \wedge b_{22} \rightarrow h_2)$

Unknown target theory:

$(b_{11} \wedge b_{12} \wedge b_{13} \rightarrow h_3) \wedge (b_{21} \wedge b_{22} \rightarrow F)$

Heads should not be deleted or added to obtain the target theory from the initial theory. In the above target theory, head h_3 is required to be added, and head h_2 is required to be deleted. This type of revisions cannot be made by the algorithm presented in this thesis.

Copyright ©Jignesh U. Doshi 2003.

Chapter 3

Related Work

This chapter contains detail notes on the work from other authors that is related to my work. I explain Angluin, et al.'s work on learning Horn formulas [2]. This work will help understand some basic concepts, which are used in Goldsmith, et al.'s work [6]. After this, I present Goldsmith, et al.'s work on revising Horn formulas and some issues with it. My work is an extension of the work already done in [6].

An algorithm for learning, not revising, Horn formulas is presented in [2]. This algorithm uses equivalence queries and membership queries to determine an unknown formula. This algorithm requires polynomial (of degree greater than 0) time in the number of variables and the number of clauses in the unknown formula. This requirement makes it unsuitable for revising Horn formulas having many more variables than clauses or errors.

An algorithm that revises Horn formulas is presented in [6]. This algorithm also uses equivalence queries and membership queries to revise an initial theory. This algorithm, however, revises theories requiring only deletion of variables. This algorithm is polynomial (of degree greater than 0) in the number of clauses in the formula, and is independent of the number of variables in the formula.

In this thesis, I extend the positive results for revising Horn formulas in [6] in one significant direction. My construction allows addition of new body variables in the revision process. The algorithm presented in this thesis, however, deals with a subclass of Horn formulas that I define as unique explanations (definition on page 9).

3.1 Learning Horn Formulas

The following information was collected from the paper "Learning conjunctions of Horn Clauses" [2].

In [2], an algorithm is presented for learning a class of Boolean formulas that are expressible as conjunctions of Horn clauses. This algorithm uses equivalence queries and membership queries to produce a formula that is logically equivalent to the unknown formula to be learned. The amount of time used by this algorithm is polynomial (of degree greater than 0) in the number of variables and the number of clauses in the unknown formula.

The gist of the algorithm presented in [2] is given below.

3.1.1 The Learning-Horn Algorithm [2]

The Learning-Horn algorithm by Angluin, et al. uses counterexamples presented by a teacher to learn a Horn formula from scratch. This algorithm keeps track of all the counterexamples presented by a teacher, and uses new counterexamples to refine its hypothesis. The hypothesis is initially true for all the examples, and is refined until the target Horn formula is learned.

According to the paper, every negative counterexample x violates some clause C of the target formula. Therefore body(C)$\subseteq$true(x), and head(C)$\in$false(x). Thus one approach presented in the paper is to add to the current hypothesis H all elements of the set

$$\text{clauses}(x) = \{ \; (\; \bigwedge_{v \in true(x)} v) \rightarrow z \; : \; z \in \text{false}(x) \; \}$$

whenever a new negative counterexample x is obtained.

Now, according to Angluin, et al., the new clauses may be incorrect (not implied by the target formula), or they may be correct, but too weak. Any clause that is not logically implied by the target formula will eventually be discovered when a positive counterexample is produced that does not satisfy this clause.

The problem of correct but weak clauses is more serious, according to Angluin, et al. To see that there is at least one correct clause, let $C^{'}$ be the clause from clauses(x) with the same head as C; the body set of $C^{'}$ may be much larger than the body set of C, with the result that there are numerous negative counterexamples that violate C but satisfy $C^{'}$. According to Angluin, et al., an adversarial choice of counterexamples can force this

approach to add exponentially many correct but weak clauses. To counter this problem, a second approach presented is to find smaller bodies by using membership queries to set more of the variables to F in the negative counterexamples that are given.

Given a negative counterexample x, set some variable that is currently T in x to F, and ask whether the result satisfies the target formula . If not, then the result still violates some clause of the target formula, and so leave the variable set to F; otherwise, set the variable back to T. Repeat this process until no more variables can be set to F.

The problem with this approach lies in the fact that even though an example violated one clause of the target formula, this minimization might produce an example that violated some other clause of the target formula; and this fact might lead to nontermination.

The first scenario shows that reduction of the number of variables set to T in the negative counterexamples is required. The second scenario rules out the greedy approach. A data driven approach is thus used by Angluin, et al. A new negative example is used in an attempt to "refine" previously obtained negative examples by intersection. Each such intersection, if it contains fewer true variables than the previously obtained negative example, is then tested to see whether it is negative. If so, this new negative example is a candidate to refine the previously obtained negative example. This algorithm maintains a sequence S of negative examples. Each new negative counterexample either is used to refine one element of S, or is added to the end of S.

In order to learn all of the clauses of the target formula, one would like the clauses induced by the (negative) examples in S to approximate distinct clauses of the target formula. This distinct-clause approximation will happen if the examples in S violate distinct clauses of the target formula. Angluin, et al. proved that an overzealous refinement may result in several examples in S violating the same clause of the target formula. To avoid this, whenever a new negative counterexample could be used to refine several examples in the sequence S, only the first among these is refined. This technique is also used in my algorithm.

I now present an example run of the algorithm from the paper [2].

3.1.2 Example Run of the Learning-Horn Algorithm

Let the variable set be $V = a, b, c, d$ and the target be $(a \wedge c \rightarrow d) \wedge (a \wedge b \rightarrow c)$. Initially, set S (a set of negative counterexamples) to an empty sequence, and H (hypothesis) to null

(true for all examples):

$$S : [\,] \text{ and } H : \phi.$$

Let the first counterexample for H be TTTF (negative example). There are no elements of S that can be refined with this negative example, so simply append this example to the end of the sequence. Since S has changed, generate a new hypothesis H.

$$S : [\text{TTTF}] \text{ and } H : (a \wedge b \wedge c \rightarrow d) \wedge (a \wedge b \wedge c \rightarrow F).$$

Let the next counterexample for H be TTTT (positive example). This eliminates an incorrect clause from H but does not change S, so don't generate a new H from S.

$$S : [\text{TTTF}] \text{ and } H : (a \wedge b \wedge c \rightarrow d)$$

All incorrect clauses from H will be discarded using future positive counterexamples in the way shown above.

Let the next negative counterexample be TTFT. Intersect this with the first element of S, and get the example TTFF, which has strictly fewer variables set to T than the first element of S had. Now, a membership query with TTFF says that this example is also a negative example, so replace the first element of S with the result of the intersection. Then, because S has changed, generate a new hypothesis H from S.

$$S : [\text{TTFF}] \text{ and } H : (a \wedge b \rightarrow c) \wedge (a \wedge b \rightarrow d).$$

The next negative counterexample is TFTF. Intersect this with the first element of S to get the example TFFF, which a membership query shows to be a positive example for the target formula. So don't refine the first element of S with TFTF, but instead add this example to the end of S.

$$S : [\text{TTFF, TFTF}] \text{ and } H : (a \wedge b \rightarrow c) \wedge (a \wedge b \rightarrow d) \wedge (a \wedge c \rightarrow d).$$

Now, the final equivalence query for H says that the target formula is learned, so stop.

3.1.3 Issues with the Learning-Horn Algorithm

This section explains some of the key issues with Angluin, et al.'s Learning-Horn algorithm [2]. The issues presented here are to some extent tackled in Goldsmith, et al.'s

work [6]. Before I go into details of Goldsmith, et al.'s work, I explain some problems with Angluin, et al.'s Learning-Horn algorithm.

The main issue with Angluin, et al.'s Learning-Horn algorithm [2] is its complexity. The amount of time used by this algorithm is polynomial (of degree greater than 0) in the number of variables and clauses in the unknown formula. Therefore, even if there are very few clauses but many variables in the target theory, this algorithm may take a long time to learn.

Also, in many cases a learner has a rough idea of what it is learning. This information is never used by Angluin, et al.'s Learning-Horn algorithm [2]. Thus Angluin, et al.'s Learning-Horn algorithm is unsuitable for learning Horn formulas where a learner has a rough idea of what it is learning, and where there are many variables in the theory that is being learned.

Goldsmith, et al.'s work on theory revision in the Deletions-Only model addresses these issues to some extent. Instead of learning Horn formulas, Goldsmith, et al.'s algorithm revises a rough estimate of what is to be learned. This algorithm is polynomial (of degree greater than 0) in the number of clauses in the formula and independent of the number of variables in the formula. The reason I say Goldsmith, et al.'s revision algorithm addresses these issues to some extent is because that algorithm revises only those theories that require deletion of variables already in the rough estimate.

3.2 Revising Horn Formulas in the Deletions-Only Model

The following notes were prepared from the paper "More theory revision with queries" [6].

In [6] an algorithm for revising Horn formulas is presented. This work is closest of all the work on learning/revising of Horn formulas to the work presented in this thesis. Instead of learning a Horn formula from scratch, revising a roughly correct Horn formula can be achieved with algorithms of lower complexity. The algorithm presented in [6] works for Horn formulas that require only deletion of variables from its clauses.

3.2.1 Deletions-Only Revision Algorithm [6]

Below I present a summary of routines used in [6] to achieve revision in the Deletions-Only model. Before that I mention some specific definitions from the paper [6].

Definitions from the paper [6]:

x $\dot{\bigcap}$ y: An example x $\dot{\bigcap}$ y is the same as $x \bigcap y$ except when there is one or more hypothesis clause C such that $x \bigcap y$ covers body(C), and x has a 1 in the position head(C), in which case that 1 stays on regardless of y.

Metaclause: A collection of all Horn clauses that have the same body.

Algorithm 1 ReviseHorn.

This algorithm routine is the main routine that starts with an empty conjunction (i.e., everything is classified as true) and repeatedly makes equivalence queries until done. For negative counterexamples, if possible the algorithm uses it to edit the body of a metaclause in the current hypothesis. The subroutine ShrinkBody that does this body variable edition is explained next. If it is not possible to shrink any of the current hypothesis clauses with the counterexample, then the algorithm uses that counterexample to add a new metaclause to the hypothesis. The subroutine NewMetaClause makes this metaclause addition. Positive counterexamples are always used to edit heads of existing metaclauses. This procedure is similar to how Angluin, et al.'s Learning-Horn algorithm [2] edits the set S of negative counterexamples.

Algorithm 2 ShrinkBody

This subroutine is the authors' first attempt to revise the hypothesis and is always called first for a negative counterexample. If this subroutine fails to shrink any of the hypothesis body, the authors call NewMetaClause (explained below). This subroutine shrinks the body of the hypothesis clause corresponding to the target clause being falsified by the given negative counterexample. If no such clause is present this subroutine fails.

Algorithm 3 NewMetaClause

This subroutine adds a new metaclause to the hypothesis and is called only when Shrink-Body fails. Next, I present some details about this subroutine, as this will help understand similar steps taken in my work.

Certain negative counterexamples can be used to create a new metaclause, because every negative instance falsifies some target clause. If negative counterexample x falsifies the target clause C^* that is a revision of some initial theory clause C_0, then $x \bigcap$ body(C_0)

also falsifies C^*. Thus, for each clause C_0 of the initial theory, one would like to say that if MQ($x \bigcap \text{body}(C_0)$) = 0, then set x to $x \bigcap \text{body}(C_0)$.

However, there are two issues to which one must pay careful attention.

First, one doesn't want to rediscover any metaclauses already in the hypothesis. Perhaps example x falsified only the target clauses not falsified by any metaclause body in the hypothesis, but $x \bigcap \text{body}(C_0)$ falsifies some target clause that is falsified by a metaclause body already in the current hypothesis.

Second, one must make sure that the process of intersecting x with the initial theory clause bodies does not change x from an example that the current hypothesis classifies as positive to one the current hypothesis classifies as negative. This is why Goldsmith, et al. use $x \dot{\bigcap} C_0$ instead of $x \bigcap C_0$ in membership query.

Algorithm 4 FixHeads

A positive counterexample x falsifies a hypothesis clause of the form ($foo \rightarrow$ F) or with a head. In the first case, Goldsmith, et al. add as heads of the metaclause body foo all heads that occur in the clauses of the initial theory ρ_0 and do not occur in foo.

In the second case, Goldsmith, et al. delete the extra head that conflicts with the counterexample. The resultant hypothesis will thus satisfy the given example; that is, the given positive counterexample will no longer be a counterexample.

Goldsmith, et al.'s algorithm revises a Horn sentence containing m clauses and needing e revisions using $O(m^3e)$ queries. Below I present an example run of their Deletions-Only revision algorithm.

3.2.2 Example Run of the Deletions-Only Revision Algorithm

Consider the following example.
Initial theory $\emptyset$: ($x_1x_2x_3 \rightarrow x_4$) $\wedge$ ($x_2x_4 \rightarrow x_5$)
Target theory $\emptyset^*$: ($x_1x_3 \rightarrow x_4$) $\wedge$ ($x_4 \rightarrow x_5$)
h = true for all examples.

Suppose our first equivalence query results in the following negative counterexample.
EQ(h) = 00010 (negative counterexample)

Since there is no hypothesis clause to shrink, we call NewMetaClause to add a new hypothesis clause. This results in the following hypothesis.

h = $x_4 \rightarrow$FALSE

Say our next example returned from an equivalence query is the following positive counterexample.

EQ(h) = 00011 (positive counterexample)

This results in a call to FixHeads. Our hypothesis now becomes,

h = $x_4 \rightarrow x_5$

Say our next example is EQ(h) = 10101 (negative counterexample)

ReviseHorn tries the following.

body($x_4 \rightarrow x_5$) $\bigcap$ 10101 = 00010 $\bigcap$ 10101 = 00000

Since MQ(body($x_4 \rightarrow x_5$) $\bigcap$ 10101) = 1, ShrinkBody is not called.

Instead we call NewMetaClause(10101, initial theory, h), which tries the following.

b = 10101 $\dot{\bigcap}$ 11100 = 10100

MQ(b) = 0

b $\bigcap$ 00010 = 00000

MQ(00000) = 1

b=10101 $\dot{\bigcap}$ 01010 = 00000

MQ(b) = 1

Thus NewMetaClause results in the following hypothesis.

h = ($x_1x_3x_5 \rightarrow$FALSE) $\wedge$ ($x_4 \rightarrow x_5$)

Next, say we get the following counterexample.

EQ(h) = 10111 (positive counterexample)

Since it is a positive counterexample, we call FixHeads. The hypothesis now becomes.

h = ($x_1x_3x_5 \rightarrow x_4$) $\wedge$ ($x_4 \rightarrow x_5$)

Our next example is EQ(h)=10101.

body($x_1x_3x_5 \rightarrow x_4$) $\bigcap$ 10101 = 10101 $\bigcap$ 10101

MQ(10101) = 0. This is a negative counterexample and so we call ShrinkBody.

ShrinkBody($x_1x_3x_5 \rightarrow x_4$,10101, ..) tries the following.

b = 10101 $\dot{\bigcap}$ 11100 = 10100

MQ(b) = 0

x = 10100

b = 10100 $\dot{\bigcap}$ 01010 = 00000

MQ(b) = 1

x = 10100

Thus ShrinkBody succeeds and results in the following.

body($x_1x_3x_5 \rightarrow x_4$) = 10100 $\bigcap$ 10101 = 10100

h = ($x_1x_3 \rightarrow x_4$) $\wedge$ ($x_4 \rightarrow x_5$)

EQ(h) = "Correct"

Thus we have revised our initial theory.

3.2.3 A question on the Deletions-Only Algorithm

Why start with an empty hypothesis, and not with a hypothesis that is the same as the initial theory in the Deletions-Only model?

I spent some time trying to answer this question. The problem with using the initial theory as our hypothesis initially is the complexity restriction. We can in fact use the above Deletions-Only algorithm with slight modifications to work with the hypothesis the same as the initial theory at the start, **but not without increasing its complexity**. The following example will make this clear.

Initial theory φ: $abc \rightarrow d \bigwedge bd \rightarrow a$

Target theory φ^*: $bc \rightarrow d \bigwedge b \rightarrow a$

Hypothesis h = $abc \rightarrow d \bigwedge bd \rightarrow a$ (Initially the same as the initial theory instead of being empty)

EQ($abc \rightarrow d \bigwedge bd \rightarrow a$)?

negative counterexample: 0110

Using ShrinkBody($abc \rightarrow d$, 0110)

h = $bc \rightarrow d \bigwedge bd \rightarrow a$

EQ($bc \rightarrow d \bigwedge bd \rightarrow a$)?

negative counterexample: 0100

This negative counterexample can be used to shrink any of the clause bodies in the hypothesis above. We don't know which one to select, so we shrink both of them. Shrinking any one of these clause bodies might result in a wrong target theory.

h = $b \rightarrow cd \bigwedge b \rightarrow ad$

h $= b \rightarrow acd$

Now the above clause can be reduced to $b \rightarrow a$ using ShrinkBody. However, we will have to reintroduce the first clause, which we already refined. In this new first clause, extra variables might appear in its body, which might be removed one at a time later. This problem makes the query complexity dependent on the number of variables in the theory. Thus the complexity of the Deletions-Only algorithm increases if we don't start with an empty hypothesis.

An extension of the work in [6] would be to allow clause body variable additions to the initial theory during the revision process and within a certain time complexity. My work in this thesis does exactly the same: It allows clause body variable additions to the initial theory clauses for a subclass of Horn formulas and achieves the revision with complexity that is logarithmic in the number of variables.

Copyright ©Jignesh U. Doshi 2003.

Chapter 4

Algorithm Overview

This chapter mentions an approach towards designing a revision algorithm in the Additions-Also model — a model that allows clause body variable additions to the initial theory in the revision process. This chapter then specifies the notation used to present my work in this thesis. An outline of the algorithm is presented next. The details of my algorithm are presented in the next chapter (Chapter 5).

4.1 Steps toward the Additions-Also Algorithm

The first step towards designing an Additions-Also algorithm would be to design an algorithm that works with a subclass of Horn formulas in which we can identify required additions. To make the problem tractable, let's assume additions are required only in bodies, and heads are unique. In order to correctly revise this type of initial theory, an Additions-Also algorithm must determine:
(a) Which variable needs to be added.
(b) To which initial theory clause this required variable needs to be added.
Once the required additions and the corresponding clauses are determined correctly, negative counterexamples can be used to add new clauses in the hypothesis. When this new hypothesis clause (associated with a unique initial theory clause requiring body variable additions) is added, all the variables required to be added to the associated initial theory clause should be present in it. This way we can successfully convert an Additions-Also problem into a Deletions-Only problem, which we know how to solve from Section 3.2.

Before I present more details of my work, let us look at the notation I use to explain

my work.

4.2 Notation

In this thesis I use the following convention.

Target theory symbols have $*$ as superscripts.

For example, ρ^* is a target theory, and C^* is a target theory clause.

Initial theory symbols have 0 as superscripts.

For example, ρ^0 is an initial theory, and C^0 is an initial theory clause.

Hypothesis theory symbols have *no* superscripts.

For example, h is a hypothesis theory, and C is a hypothesis clause.

I use x, b for examples or temporary variables, and $-$ for asymmetric set difference.

4.3 Outline

Using the initial theory, the algorithm constructs a hypothesis starting from an empty Horn formula (which is always satisfied). Each clause I add to the hypothesis:

- revises an initial theory clause, and
- covers a target clause with the same head as that of the initial theory clause being revised.

Furthermore, as I show in Lemma 6.5, a hypothesis clause is a subset of an initial theory clause union the corresponding target theory clause. In other words, to each new clause added to the hypothesis, no unnecessary literals are added other than those already in the initial theory clause being revised.

The other task of my construction is to remove all the unnecessary literals from the hypothesis clauses. While Goldsmith, et al. have presented an algorithm for revision of Horn theories that require deletions only [6], mine differs from theirs in two ways. First, the addition and deletion steps are mixed in my construction. It may be that I perform deletions on some hypothesis clauses before adding others. Second, because of my assumptions about

the clause heads (implications) being unique, un-erasable and disjoint from other clause bodies, the deletion step in my algorithm is much simpler than in [6].

My construction is governed by the algorithm ReviseHorn(ρ^0). Within ReviseHorn(ρ^0) are calls to ShrinkBody($\ddot{x}$, C^0) for deletions of unnecessary variables in the hypothesis clauses, and to Associate(x, ρ^0), which begins the process of adding a new clause to the hypothesis. Associate(x, ρ^0) is so named because this subroutine associates a particular head, and thus a particular initial theory clause, with a counterexample. Once this association is done, Associate(x, ρ^0) calls BinarySearch($\ddot{x}, C^0$) to find any necessary additions to an associated initial theory clause. More formally,

Algorithm 1: ReviseHorn(ρ^0) revises a given Horn formula ρ^0. This subroutine removes unnecessary variables from the body of any hypothesis clause if possible, or else adds to the hypothesis an entire new clause that has all the required body variables.

Algorithm 2: ShrinkBody(x, h) removes unnecessary variables from a hypothesis clause body if possible and indicates whether this shrinking was successful or not.

Algorithm 3: Associate(x, ρ^0) finds an association between a given negative counterexample x and some initial theory clause. This association is then used by BinarySearch($\ddot{x}$, C^0) to find additions of new body variables, if any are in fact required.

Algorithm 4: BinarySearch($\ddot{x}$, C^0) uses negative counterexample $\ddot{x}$ to find all missing body variables in the initial theory clause C^0 and add them to the new hypothesis clause body.

Goldsmith, et al. based their Horn Revision algorithm [6] on Angluin et al.'s algorithm for learning Horn formulas via queries [2]. I model my algorithm loosely on that of Goldsmith, et al. For instance, my BinarySearch($\ddot{x}, C^0$) is similar to that presented in [6] except that their BinarySearch always finds a necessary addition, whereas mine first checks whether any additions are required. ShrinkBody(x, h) is also similar to that in [6], but Associate(x, ρ^0) does not appear in any of the surveyed literature.

Copyright ©Jignesh U. Doshi 2003.

Chapter 5

Algorithms

This chapter provides commented pseudo–code along with a line-by-line explanation of all the subroutines of the Additions–Also revision algorithm. Chapter 6 uses this pseudo–code to prove the correctness of lemmas, which are later used to prove the correctness of my algorithm.

5.1 ReviseHorn(ρ^0)

Revises a Horn formula ρ^0 to ρ^*.
Input: ρ^0 is the initial theory to be refined.
We cannot get a positive counterexample in the entire revision process. This fact is proved in Lemma 6.7.

1: h=Empty Horn formula (always satisfied)

2: **while** ((x=EQ(h)) $\neq$ "Correct") **do**
// By lemma 6.7, x will always be a negative counterexample.

3: $shrunk$=ShrinkBody(x, h)

4: **if** (**not** $shrunk$) **then**

5: h=$h \wedge$ Associate(x, ρ^0) // Add a new hypothesis clause.

6: **end if**

7: **end while**

ReviseHorn(ρ^0) begins with an empty hypothesis (i.e., everything is classified as *true*) and repeatedly makes equivalence queries until done. For every negative counterexample x, if possible I use x to edit the body of a clause in the current hypothesis, or else I use x to add a new clause to the hypothesis. The second case will always occur with the first negative counterexample. By Lemma 6.7, positive counterexamples are never obtained.

The call to ShrinkBody(x, h) on line 3 is used to edit an existing hypothesis clause body if possible. If shrinking is not possible, I call Associate(x, ρ^0) to add a new clause in the hypothesis. This new clause has all the required body variables added to it.

5.2 ShrinkBody(x, h)

Shrinks a hypothesis clause if possible and indicates whether shrinking was done or not.
Input: negative counterexample x, hypothesis theory h.

1: **for** (each clause $C \in h$ in order) **do**
// Check if any hypothesis clause can be shrunk.

2: **if** (body(C)$\cap x \subset$ body(C) **and** MQ($x \cap$ body(C) $\cup$ (all heads) $-$ head(C))==0) **then**

3: body(C) = body(C)$\cap x$

4: **return** *true* // Shrinking is done.

5: **end if**

6: **end for**

7: **return** *false* // No shrinking done.

In lines 1 and 2, I check whether any of the current hypothesis clause bodies can be shrunk. If so, I shrink them in line 3 and return *true* indicating that shrinking was done.

In fact, there may be a target clause C^* such that x covers both body(C^*) and head(C^*), but x could be used to remove unnecessary variables from the body of the corresponding hypothesis clause – even though x is not a counterexample for that clause. This is why, in line 2 of ShrinkBody(x, h), I turn *on* all heads except the head of the clause under consideration. Turning *on* the other heads guarantees that a negative membership query response can only reflect that body(C^*) is covered. If shrinking is not possible, it is indicated by returning *false* in line 7.

5.3 Associate(x, ρ^0)

Associates a negative counterexample to an initial theory clause. This association is then used by BinarySearch($\ddot{x}, C^0$) to find all the required additions to this associated initial theory clause.

Input: x is a negative counterexample.

1: $\ddot{x}$=$x \cup$ all heads of ρ^0 // $\ddot{x}$ is not a counterexample at this point.

2: d=$\ddot{x} - x$ // all heads in $\ddot{x}$ not already *on* in x.

3: **while** (MQ($\ddot{x}$)==1) **do**

4: Turn *off* a bit b in d such that $\forall\ C \in h$, $b \notin$ head(C)

5: $\ddot{x}$=$x - b$

6: **if** ($\ddot{x}$ is a negative counterexample) **then**

7: Associate $\ddot{x}$ to clause C^0 of ρ^0 such that head(C^0)==bit b in d that was just turned *off*

8: **end if** // By Lemma 6.7, $\ddot{x}$ cannot become a positive counterexample.

9: **end while**

10: $\ddot{x}$=**BinarySearch(**$\ddot{x}$**,** C^0**)** // finds all required additions if any in C^0.

11: **return** clause: $\ddot{x} \rightarrow$ head(C^0)

In line 1, I turn *on* (set to *true*) all the heads that were *off* initially in the negative counterexample x. I call this new example $\ddot{x}$. Since $\ddot{x}$ has all the possible heads turned *on*, it can never be a counterexample at this step. All heads turned *on* means $\ddot{x}$ satisfies all the clauses of the target and all the clauses of the hypothesis theory. As there is no clause with head 'F', $\ddot{x}$ will be positive for both the target and the hypothesis theory.

In line 2, I use d to keep track of all those heads that are *on* in $\ddot{x}$ and *off* in x. Associate(x, ρ^0) is called only when shrinking is not possible, that is, when a new hypothesis clause corresponding to a target clause falsified by x needs to be added to the hypothesis. The head of this target clause will be *off* in x, and this head will also not be the head of any current hypothesis clause. Thus d contains at least one head b that is not a head of any current hypothesis clause such that turning b *off* in $\ddot{x}$ makes $\ddot{x}$ a negative counterexample.

In the *while* loop, I find the initial theory clause to which $\ddot{x}$ needs to be associated before calling BinarySearch($\ddot{x}$, C^0). BinarySearch($\ddot{x}$, C^0) then finds all the body variable additions required for the associated initial theory clause.

At line 3, I am certain that $\ddot{x}$ is positive for both the initial theory and the target theory. In lines 3..9, I turn *off* one head at a time from d in $\ddot{x}$ and check whether $\ddot{x}$ becomes a negative counterexample.

In line 5, one of the heads that was *on* in $\ddot{x}$ and *off* in x is turned *off* in $\ddot{x}$ using d. I don't turn *off* any existing hypothesis head, as I don't want to associate this new negative counterexample with an existing hypothesis clause.

In line 6, I check if $\ddot{x}$ is turned into a negative counterexample due to turning *off* of a head in previous line. In line 7, I associate the negative counterexample $\ddot{x}$ with the initial theory clause having the head that was just turned *off*. Having found an association, I exit the loop. I do not check if $\ddot{x}$ is a positive counterexample because, by Lemma 6.7, I cannot get a positive counterexample in the entire revision process. In line 9, I loop back to line 3 keeping the head turned *off* in $\ddot{x}$ as no association was found.

In line 10, I call BinarySearch($\ddot{x}$, C^0) to find all required additions, if any, to the associated clause C^0 before returning $\ddot{x}$ to ReviseHorn(ρ^0) on line 11.

5.4 BinarySearch($\ddot{x}, C^0$)

Finds all the required body variable additions, if any.

Input: $\ddot{x}$ is a negative counterexample associated with C^0

1: $start = x = (\ddot{x} \cap body(C^0)) \cup$ all head bits in $\ddot{x}$

2: **while** $(\ddot{x} - x \neq 0)$

3: $\quad d = \ddot{x} - x$

4: $\quad$ **repeat**

5: $\quad\quad$ **if** (it's the first iteration of the repeat loop) **then**

6: $\quad\quad\quad c = \phi$ // First check whether any additions are needed.

7: $\quad\quad$ **else**

8: $\quad\quad\quad c = d$ with half of its *on* bits turned *off*

9: $\quad\quad$ **end if**

10: $\quad\quad$ **if** $(\text{MQ}(x \cup c) == 0)$ **then**

11: $\quad\quad\quad \ddot{x} = x \cup c$

12: $\quad\quad\quad d = c$

13: $\quad\quad$ **else**

14: $\quad\quad\quad x = x \cup c$

15: $\quad\quad\quad d = d - c$

16: $\quad\quad$ **end if**

17: $\quad$ **until** (number of bits *on* in $d \leq 1$)

18: $x = start = start \cup d$ // d is the required addition.

19: **end while**

20: **return** ($x-$ all head bits) // Remove all head bits.

BinarySearch($\ddot{x}, C^0$) finds all the required body variable additions (variables that were not present in the corresponding initial theory clause) to the body of the new clause being added to the hypothesis.

The clause returned by BinarySearch($\ddot{x}, C^0$) may not be the final target theory clause. This clause might require deletions, and that will be done by shrinking it with future negative counterexamples. But for now we are guaranteed that this returned clause needs no further additions.

In the first line, I temporarily keep the heads *on* in x so that clauses with these heads don't interfere with the addition checks performed later. An interference occurs when x covers bodies of these clauses (clauses whose head is turned *off* after intersection in the first line). This interference might lead to additions to the wrong clause being found. These heads are later removed from x after all additions for body(C^0) are found and before returning x as the new clause body. In line 8, it does not matter what convention is chosen for determining which half of the bits is turned *off*. The reader is invited to choose a convention.

In the repeat..until loop, each required addition is found using binary search on the possible additions. A required variable is the threshold variable that when turned *off* in $\ddot{x}$ makes $\ddot{x}$ (a negative counterexample) a non-counterexample. In line 18, I have found a required addition, which is noted and not included for examination in binary searches for further additions. The outer *while* loop performs this binary search process until no new additions are possible.

In the last line, I remove all the head variables from x because heads cannot be in the new clause body (by my assumption that theories are Unique Explanations). After this removal of heads, I return x to Associate(x, ρ^0) as the new clause body to be added to the hypothesis.

Copyright ©Jignesh U. Doshi 2003.

Chapter 6

Lemmas and Theorems

This chapter provides some lemmas and theorems with their proofs. These lemmas and theorems are used later in Chapter 7 for proving the correctness of my algorithm.

Lemma 6.1 *The first counterexample in the construction must be a negative counterexample.*

Proof Initially the hypothesis theory is empty, that is, it is true for all examples in the universe. So the only way we can have a counterexample is by having an example that satisfies the hypothesis theory and falsifies the target theory. Thus the first counterexample in the construction must be a negative counterexample. ∎

Lemma 6.2 *Given the negative counterexample x in Associate(x, ρ^0), x falsifies a target clause C_j^* such that h_j^* is not the head of any clause already present in the current hypothesis.*

Proof Suppose hypothesis $h = b_1 \rightarrow h_1 \wedge b_2 \rightarrow h_2 \wedge ... \wedge b_k \rightarrow h_k$ where b_i is the body, and h_i is the head for clause C_i in the hypothesis. Say x is a new negative counterexample. Then $h(x)$ is true. For each clause C_i in h, x either covers its body b_i and includes its head h_i=h_i^*, or it does not cover body b_i.

Next, we claim that if x does not cover body b_i it does not cover the corresponding target clause body b_i^* and hence cannot be a negative counterexample due to clause C_i^*. To prove this, let's assume x does not cover a hypothesis body b_i but covers the corresponding target clause body b_i^*, and x does not include head h_i=h_i^*. This means $b_i^* \subset b_i$, and example x is a negative counterexample. Prior to calling Associate(x, ρ^0), we attempt to shrink the hypothesis clause body b_i using x. The shrinking of b_i will succeed since $b_i^* \subset b_i$, and

so we will never call Associate(x, ρ^0). Thus x in Associate(x, ρ^0) cannot cover body b_i^* if does not cover the corresponding hypothesis clause body b_i and hence cannot be a negative counterexample due to target clause C_i^* corresponding to C_i already being present in the hypothesis. ■

Lemma 6.3 *Given a negative counterexample x to hypothesis h, Associate(x, ρ^0) produces a negative counterexample $\ddot{x}$ and head h^* such that h^* is the head of the unique target clause C^* that is not present in the current hypothesis, and $\ddot{x}$ covers body(C^*). In other words, we correctly associate each counterexample with exactly one initial theory clause and hence exactly one target theory clause not present in the current hypothesis.*

Proof Suppose hypothesis $h = b_1 \rightarrow h_1 \wedge b_2 \rightarrow h_2 \wedge ... \wedge b_k \rightarrow h_k$ where b_i is the body and h_i is the head for a clause C_i in the hypothesis. Associate(x, ρ^0) ensures that $\ddot{x}$, obtained by turning *on* all heads in x, is not associated with any of the $C_i, i \leq k$, already present in the hypothesis. This property is ensured by not turning *off* any hypothesis head $h_i \in \ddot{x}$. Initially in Associate(x, ρ^0), $\ddot{x}$ is a non–counterexample as all heads are *on* in $\ddot{x}$. This fact also makes $\ddot{x}$ true for both the target theory and the hypothesis theory. The example $\ddot{x}$ can never become a positive counterexample as we never turn *off* any hypothesis head $h_i \in \ddot{x}$ or turn *on* any hypothesis clause body variable. In Associate(x, ρ^0) we only turn *off* heads that were turned *on* by us in $\ddot{x}$ and that are not the head of any of the existing hypothesis clauses. We stop with the first initial theory clause C_j^0 whose head when turned *off* in $\ddot{x}$ made $\ddot{x}$ a negative counterexample. At this point $\ddot{x}$ falsifies only a single target theory clause not present in the current hypothesis and having the same head as C_j^0. Thus $\ddot{x}$ is associated with C_j^0 and the corresponding target clause C_j^* not present in the current hypothesis. Associate(x, ρ^0) thus produces a negative counterexample $\ddot{x}$ and head h_j^*=head(C_j^0) such that h_j^* is the head of the unique target clause C_j^* not present in the current hypothesis, and $\ddot{x}$ covers *body*(C_j^*). ■

Lemma 6.4 *Given $\ddot{x}$, h^*, and an initial theory clause $C^0 = b^0 \rightarrow h^*$ corresponding to a target theory clause $C^* = b^* \rightarrow h^*$, BinarySearch($\ddot{x}$, C^0) produces all and only variables in $b^* - b^0$. More simply, BinarySearch($\ddot{x}$, C^0) finds all the required clause body additions for an initial theory clause before it is added as a new hypothesis clause.*

Proof In line 1 of BinarySearch($\ddot{x}$, C^0), we temporarily add all the heads that are *on* in the counterexample $\ddot{x}$ to the new hypothesis clause body. This addition of heads prevents

interference of other target clauses in the addition check; that is, it prevents BinarySearch($\ddot{x}$, C^0) from finding required additions to clauses other than C^0. In the *repeat-until* loop of BinarySearch($\ddot{x}$, C^0), we find a single variable addition, if any, to body(C^0) that is the threshold between a positive and a negative example associated with clause head(C^0). In the outer loop of BinarySearch($\ddot{x}$, C^0), we perform this binary search repeatedly to find each required addition. This search continues until no more additions are required. Thus BinarySearch($\ddot{x}$, C^0) finds all the required clause body additions for an initial theory clause before this clause is added as a new hypothesis clause. ∎

Corollary 6.5 *The clause resulting from the binary search in Lemma 6.4, $C = b \rightarrow h^*$, corresponds to the target clause $C^* = b^* \rightarrow h^*$, and $b^* \subseteq b$. Furthermore, $b - b^* \subseteq b^0$, that is, the only unnecessary body variables in this new hypothesis clause are the ones already present in the initial theory clause body.*

Proof In BinarySearch($\ddot{x}$, C^0) we remove all the heads from the body to be returned. These heads were added by Associate(x, ρ^0) and kept *on* in BinarySearch($\ddot{x}$, C^0) during the search. By Lemma 6.4, BinarySearch($\ddot{x}$, C^0) adds only necessary variables to the body being returned. The only unnecessary variables that are allowed in the body that is returned are those that were present in the initial theory body and in the counterexample.These variables are later removed by ShrinkBody(x, h). Thus BinarySearch($\ddot{x}$, C^0) makes all the required additions and no unnecessary additions. Since we make all the required additions prior to adding a new clause, the clause resulting from the binary search in Lemma 6.4, $C = b \rightarrow h^*$, corresponds to the target clause $C^* = b^* \rightarrow h^*$, such that $b^* \subseteq b$. ∎

Lemma 6.6 *If ShrinkBody(x, h) is applied to $b \rightarrow h^*$, it will remove only variables in $b - b^*$. In other words, ShrinkBody(x, h) deletes variables from hypothesis clauses only if those variables do not appear in the corresponding target clause.*

Proof From Lemma 6.5 each hypothesis clause has a corresponding target clause, and this association is never changed during the revision process. In line 2 of ShrinkBody(x, h), we check whether removal of at least one hypothesis body variable not present in the corresponding target clause makes the negative counterexample a non–counterexample. The temporary turning *on* of other heads in the second condition guarantees that even after deletion of variables from the hypothesis clause C, body(C) covers the same target clause

body as previously associated. Thus ShrinkBody(x, h) deletes variables from hypothesis clauses only if those variables do not appear in the corresponding target clause. ■

Corollary 6.7 *All counterexamples are negative; that is, we can never get a positive counterexample in the revision process. In other words, the target theory always implies the hypothesis.*

Proof This follows from Corollary 6.5 and Lemma 6.1. Initially the hypothesis is true everywhere, so the first counterexample is always negative. By Lemma 6.5, the clauses we add to the hypothesis are each associated with a unique target clause having the same head. At any point in the revision process each hypothesis clause body has all the required additions, and from Lemma 6.6, ShrinkBody(x, h) removes body variables from a hypothesis clause only if they are not present in the corresponding target theory. So each hypothesis clause body is always a superset of the corresponding target clause body having the same head. Thus the target theory always implies the hypothesis theory. Therefore we cannot have an example that is true for the target theory and false for the hypothesis theory. Thus all counterexamples are negative; that is, we can never get a positive counterexample in the revision process. ■

Copyright ©Jignesh U. Doshi 2003.

Chapter 7

Correctness Analysis

This chapter analyzes the correctness of my algorithm using the lemmas and theorems from Chapter 6. A derivation for the complexity of my algorithm is presented at the end of this chapter to determine the maximum number of membership queries and equivalence queries required for correct revision.

Let m be the number of clauses in the target theory, n be the total number of variables, and e be the revision distance from the target theory. I first prove the correctness of my algorithm and then address its complexity.

Lemma 7.1 *ReviseHorn(ρ^0) correctly revises a Horn formula ρ^0 with m clauses and n variables and with revision distance e from the target theory ρ^* (where ρ^0 and ρ^* are Unique Explanations).*

Proof Once a particular clause is added, it is never deleted. By Lemma 6.6, each nonempty hypothesis is implied by ρ^*. In particular, by Lemma 6.5, each hypothesis clause covers the corresponding ρ^* clause (i.e., the target clause with the same head). For each variable appearing in a hypothesis clause that does not appear in the corresponding ρ^* clause, there will eventually be a counterexample that triggers ShrinkBody(x, h) on that clause.

Next, I show that each ρ^* clause is eventually covered by some hypothesis clause. Suppose not, and let C^* be a target clause with head h^* that is never covered. Then in particular the instance with all bits turned *on* except h^* will be a negative counterexample to any such hypothesis. In order for my algorithm to converge, some such counterexample must be handled.

Finally, I show that my algorithm actually converges. By Lemma 6.5, no unnecessary additions are made to any of the hypothesis clauses, so there can only be m clauses added, and there can be no more than e deletions per clause. Since each clause addition is finite, and each deletion requires only finitely many queries, my algorithm converges. ■

Let us now consider how many membership queries and equivalence queries are required for the revision process.

Lemma 7.2 *ReviseHorn(ρ^0) makes $O(m^2 + m * e * \log n)$ membership queries (MQs) to revise a Horn formula with m clauses and n variables and with revision distance e from the target theory.*

Proof In a single call to ShrinkBody(x, h), line 2 of ShrinkBody(x, h) is executed at most m times as there can be at most m hypothesis clauses. Thus each call to ShrinkBody(x, h) makes at most m MQs.

Line 5 of ReviseHorn(ρ^0) is executed exactly m times as there will be m additions of clauses to the initial empty hypothesis.

In a single call to Associate(x, ρ^0), line 3 is executed at most m times. Thus m MQs are made at line 3.

The *repeat-until* loop in a single call to BinarySearch($\ddot{x}$, C^0) is executed at most $\log n$ times, and the *while* loop is executed at most e times. Thus a single call to BinarySearch($\ddot{x}$, C^0) makes $e * \log n$ MQs. As a result, each call to Associate(x, ρ^0) makes $(m + e * \log n)$ MQs. Therefore line 5 of ReviseHorn(ρ^0) makes a total of $m * (m + e * \log n)$ MQs.

Line 3 of ReviseHorn(ρ^0) is executed at most $e + m + 1$ times for at most e required deletions and at most m shrinking calls that failed just before the addition of new clauses by Associate(x, ρ^0). Thus line 3 of ReviseHorn(ρ^0) makes a total of $m * (e + m + 1)$ MQs.

Therefore the total number of MQs made is at most $(m*(e+m+1)+m*(m+e*logn))$, which is $O(m^2 + m * e * \log n)$.

Lemma 7.3 *ReviseHorn(ρ^0) makes $O(e + m)$ equivalence queries (EQs) to revise a Horn formula with m clauses and n variables and with revision distance e from the target theory.*

Proof The only place equivalence queries are used in the algorithm is in line 3 of ReviseHorn(ρ^0) and since line 3 of ReviseHorn(ρ^0) is executed at most $(e + m + 1)$ times, my algorithm will need at most $(e + m + 1)$ EQs to make the revision. ■

Thus ReviseHorn(ρ^0) correctly revises a Horn formula ρ^0 with m clauses and n variables and with revision distance e from the target theory ρ^* (where ρ^0 and ρ^* are Unique Explanations) using at most $O(m^2 + m * e * \log n)$ MQs and $O(e + m)$ EQs.

Copyright ©Jignesh U. Doshi 2003.

Chapter 8

Implementation

This chapter covers all the implementation issues of the revision application and presents information regarding the application design, the application usage, its inputs and outputs, the data structures used and the validations performed. Section 8.1 explains why Java was chosen to implement the algorithm. This section mentions important Java APIs used in the application development. Section 8.2 presents a high-level design of the application along with a class diagram. Section 8.3 explains what are mandatory and optional inputs to the application and how the inputs such as the universe of variables, the initial theory and the target theory are entered in the application. Section 8.4 specifies the data structures used to represent clauses and formulas in the application. Section 8.5 mentions other required data that are derived from the user inputs such as the set of valid head variables and the set of valid body variables. Section 8.6 explains what validations are required and how clause validations and theory validations are performed. Section 8.7 explains how equivalence query (EQ) and membership query (MQ) functions are implemented. Section 8.8 explains how the revision functions are implemented. Section 8.9 lists all the types of output generated by the application. The application code is presented in the appendix.

8.1 Programming Language

I used Java to develop the revision application. The primary reason for choosing Java was availability of the HashSet class in Java 2 platform API specification. The HashSet class API allows some basic set operations such as intersection, union and set difference to be performed efficiently on HashSet objects. Use of the HashSet class API considerably

reduced programming time and effort. Java also helped in developing a better GUI for the revision application.

8.2 Design

The application development involved designing the classes (to hold the data and represent the objects) and the interfaces (interaction among these objects). The following objects were designed.

8.2.1 Clause object

This object was designed to hold a Horn clause along with the operations that can be carried out on a Horn clause. This object contains a variable set (a clause body) and a variable (a clause head) representing a Horn clause. This object has two constructors, one to construct a clause using variables and the other to construct a clause from another clause object. This object also provides functions to set body variables and get body and head variables. Notice that no function to set a head is provided because in the revision process a clause head is never changed.

8.2.2 Global object

This object contains all the global variables used by ReviseHorn object and the GUI object. This object contains the initial theory, the hypothesis theory and the target theory along with functions used to display them in the format understood by the user. This object also implements membership query and equivalence query functions described in section 8.7. This object contains other global variables such as the counts for membership and equivalence queries.

8.2.3 ReviseHorn object

This object contains all the revision functions (ShrinkBody, Associate and Binary-Search) and other functions to get the theories from a user and validate these theories. This object uses global variables from the Global object to store and retrieve data during the revision, gets inputs from the GUI object and displays output through the GUI object.

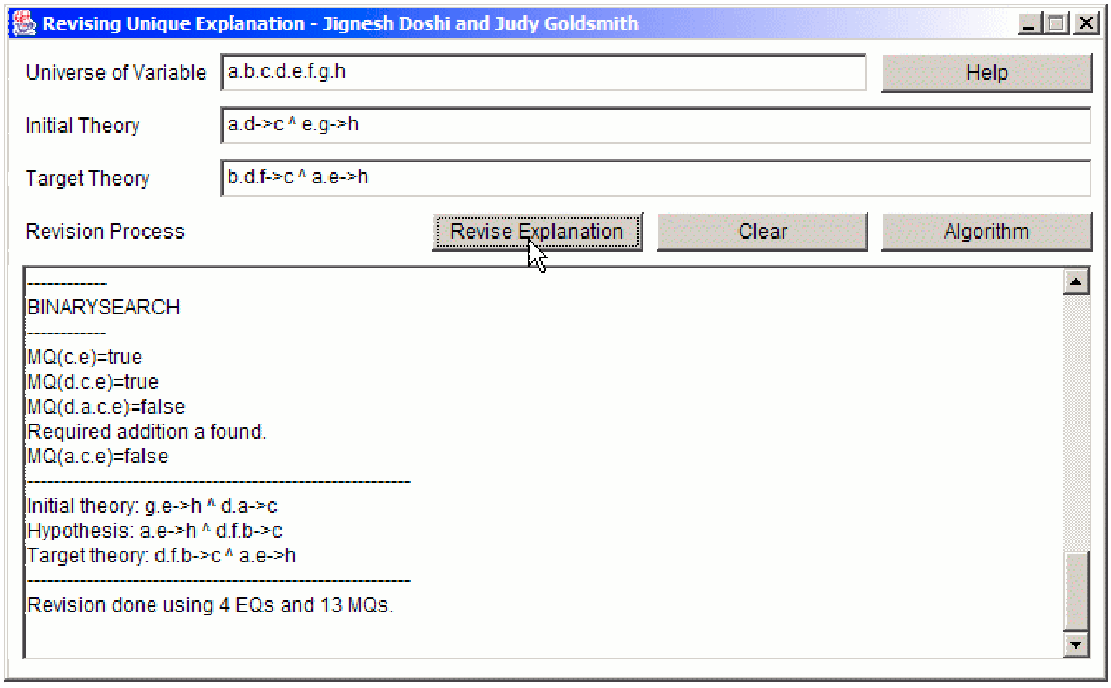

Figure 8.1: Application GUI

8.2.4 GUI object

This object provides an interface between the user and the revision application. This object takes inputs from the user and passes these inputs to the ReviseHorn object for processing, or to the Global object for storage. All the actions are initiated via this object. This object also provides help information and links to a technical paper on the algorithm. A snapshot of the application GUI is shown on page 40.

The diagram on page 41 shows all the objects designed to realize the revision application along with the interactions between them.

8.3 Inputs

This section describes all the inputs to the application and how they are entered.

8.3.1 Universe of variables

The universe of variables is a set of all variables present in the initial theory and the target theory. This is entered in the application as, for example, $a.b.c.d.e.f.g$ where

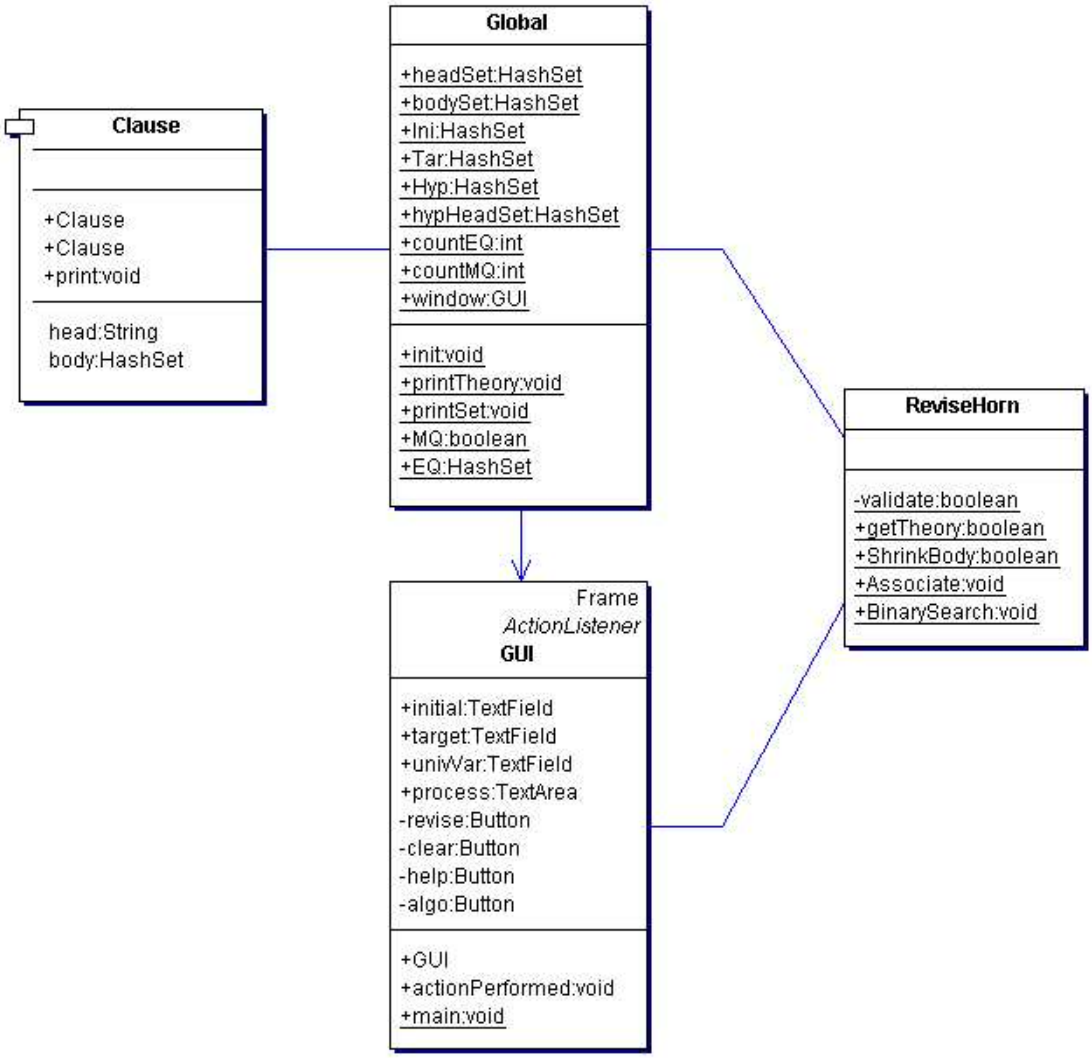

Figure 8.2: Application Class Diagram

a, b, c, d, e, f and g are strings representing Boolean variables. This input is mandatory.

8.3.2 Initial theory

The initial theory is the next required input. All the clauses are separated by ' $\wedge$ ' representing a Boolean AND. Clause body variables are separated by '.' and are entered before a clause head. Clause body and clause head are separated by '$\rightarrow$' symbol.
Each Horn clause is entered as, for example, $a.b.c.d \rightarrow e$.
The initial theory is entered as, for example, $a.b.c \rightarrow d \wedge c.e \rightarrow f \wedge a.g \rightarrow h$.

8.3.3 Target theory

The target theory is a mandatory input, which is entered the same way the initial theory is entered.

8.4 Data Structures

The data structures designed were mainly to hold Horn clause data, Horn theory data and the counterexamples. This section provides details about all the data structures designed for the revision application.

8.4.1 Clause

A clause object as discussed earlier is used to store a Horn clause data. A **clause** object contains the following.

- A **String** object for storing the head information. So, for example, a head can be '$WillCook$' or it can be just a character 'a'.
- A **HashSet** of **String** objects for storing the body information. So, for example, a body can be a set {*'HisTurn', 'NoLeftOvers', 'NotInvited'*} or a set {*'b', 'c', 'd', 'e'*}.

8.4.2 Theory

A theory object is used to store information about the initial theory, the target theory and the hypothesis theory. A theory is a **HashSet** object containing

- **Clause** objects, each of which represents a Horn theory clause.

8.4.3 Counterexamples

A counterexample is a **HashSet** of strings. So, for example, a counterexample can be $\{`HisTurn', `NoLeftOvers', `NotInvited'\}$ meaning variables $HisTurn$, $NoLeftOvers$ and $NotInvited$ are set to true or a set {*'b', 'd', 'e'*} meaning variables b, d and e are set to true.

8.5 Derived Inputs

I tried to minimize the inputs from the user by deriving some of the required inputs. These variables are stored within the application as:

8.5.1 Set of valid head variables

A set derived from the initial theory. This set is compared with the set of target theory heads. These sets must be equivalent. This condition is one of the requirements for the theories to be unique explanations.

8.5.2 Set of valid body variables

This set is a set difference of the universe of variables and the set of valid heads. This set is used for checking body variables in the target theory.

8.5.3 Hypothesis

A hypothesis is initially an empty Horn formula, which is true for all examples. A hypothesis is derived from the initial theory, the target theory and the counterexamples obtained from equivalence queries (EQ).

8.6 Validation

The derived inputs obtained above are used to validate the input clauses and theories. These validations are performed before the revision process starts, and the user is prompted to make corrections if any are needed.

8.6.1 Clause Validation

The main clause-related validations are the following.

- All the variables in a clause must be in the universe-of-variable set.
- A head variable must not be in the body of the same clause, as this will make the clause always true.
- All the clause body variables must be in the set of valid body variables.

- The clause head must be in the set of valid head variables.

8.6.2 Theory Validation

The initial theory and the target theory must be unique explanations. To validate this, the following must be checked.

- Clause heads are unique; that is, no two clauses in a theory have the same head.
- Clause heads are not the literal 'F'. This application does not allow literals to be entered.
- Clause heads should not be in the body of any other clause in the theory.
- A set of target theory heads must be the same as the set of initial theory heads.

8.7 Query Functions

Two query functions, membership and equivalence query functions, were implemented for the revision application. The algorithm for these functions is presented below.

8.7.1 Equivalence Query (EQ)

Inputs: A target theory and a hypothesis theory.
Returns: An empty set when the target and the hypothesis theories are equivalent and a set of variables if the target and the hypothesis are not equivalent. This set of variables is a counterexample, and it always falsifies the target and satisfies the hypothesis; that is, this set is always a negative counterexample. Positive counterexamples are never obtained in the entire revision process.
Implementation: We try to build a counterexample falsifying a target clause. This is achieved by having a set of all body variables of a target clause and all the valid head variables except the head of this target clause. This set (example) thus falsifies the target theory. We then check if this example satisfies the current hypothesis. If this example does satisfy the hypothesis, we return this example as a negative counterexample. Otherwise, we try building a negative counterexample with the next target theory clause. If no such negative counterexample could be generated, we return an empty set, which indicates that the hypothesis and the target are equivalent. Also, if a negative counterexample is

found, we add some random variables to it such that this counterexample still remains a negative counterexample. This represents uncertainty in the nature of the answers from an equivalence oracle.

8.7.2 Membership Query (MQ)

Inputs: A theory and a set of variables representing an example whose membership to the theory is being tested.
Returns: True if the example makes the given theory true; that is, the example is a member of the given theory and false if the example makes the given theory false (the example is not a member of the given theory).
Implementation: We check if the given example falsifies any of the given theory clauses. An example falsifies a clause if it covers the set of clause body variables and not the clause head. If any such clause is found return false. Otherwise, return true.

8.8 Revision Functions

The functions involved in the revision process are: ShrinkBody, Associate and Binary-Search. These functions are implemented by line-by-line translation of the corresponding pseudo–code shown earlier into Java code.

8.9 Output

The following outputs are generated by the revision application.

- All the steps taking place in the revision process are shown along with the counterexamples obtained, queries asked and the answers obtained.
- The hypothesis theory derived after obtaining a counterexample from every equivalence query during the revision process.
- The number of equivalence queries (EQs) and membership queries (MQs) asked during the revision process is calculated and shown after the revision is accomplished.

Copyright ©Jignesh U. Doshi 2003.

Chapter 9

Example Run

This chapter presents an example run of the application developed. The exact output generated by the application for an example is shown here.
Universe of variables: a.b.c.d.e.f.g.h.i.j.k.l.m.n.o
Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o
Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

EQ(h)=o.d.i.k.m.a.h.c.n.b

SHRINKBODY

No shrinking done.

ASSOCIATE

MQ(d.o.i.k.m.a.c.h.n.b.e)=true
MQ(d.o.i.k.m.a.c.h.n.b)=false
Associating with: a.c.b → e

BINARYSEARCH

MQ(o.k.a.h.c.n.b)=true
MQ(d.o.k.a.h.c.n.b)=false

Required addition d found.

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: d.a.c.b → e

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= o.d.i.k.m.h.c.n.b

SHRINKBODY

MQ(o.d.k.h.c.n.b)=false

Clause shrunk: d.c.b → e

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: d.c.b → e

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= d.i.k.m.a.h.c.n.b.e

SHRINKBODY

No shrinking done.

ASSOCIATE

MQ(o.d.i.k.m.a.c.h.n.b.e)=true

MQ(d.i.k.m.a.c.h.n.b.e)=false

Associating with: c.b → o

BINARYSEARCH

MQ(k.h.c.n.b.e)=false

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ d.c.b → e

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= d.o.i.m.a.h.c.f.n.e

SHRINKBODY

MQ(k.h.c.n.e)=true

MQ(o.d.k.h.c.n)=true

No shrinking done.

ASSOCIATE

MQ(o.d.i.k.m.a.c.h.f.n.e)=true

MQ(o.d.i.m.a.c.h.f.n.e)=false

Associating with: i.j → k

BINARYSEARCH

MQ(o.i.h.n.e)=true

MQ(d.o.i.m.h.n.e)=true

MQ(d.o.i.m.a.h.n.e)=true

MQ(d.o.i.m.a.c.h.n.e)=false

Required addition c found.

MQ(d.o.i.c.h.n.e)=false

Required addition d found.

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ d.i.c → k ∧ d.c.b → e

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= o.i.m.a.h.c.f.n.e

SHRINKBODY

MQ(k.h.c.n.e)=true

MQ(o.i.h.c.n.e)=false

Clause shrunk: i.c → k

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ i.c → k ∧ d.c.b → e

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= d.o.i.k.m.a.f.n.g.e

SHRINKBODY

MQ(k.h.n.e)=true

MQ(o.i.h.n.e)=true

MQ(o.d.k.h.n)=true

No shrinking done.

ASSOCIATE

MQ(o.d.i.k.m.a.h.f.n.g.e)=true

MQ(o.d.i.k.m.a.f.n.g.e)=false

Associating with: f.g → h

BINARYSEARCH

MQ(o.k.f.n.g.e)=true

MQ(d.o.i.k.f.n.g.e)=true

MQ(d.o.i.k.m.f.n.g.e)=true

Required addition a found.

MQ(d.o.k.a.f.n.g.e)=false

Required addition d found.

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ i.c → k ∧ d.c.b → e ∧ d.a.f.g → h

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= o.i.k.m.a.f.n.g.e

SHRINKBODY

MQ(k.h.n.e)=true

MQ(o.i.h.n.e)=true

MQ(o.k.h.n)=true

MQ(o.k.a.f.n.g.e)=false

Clause shrunk: a.f.g → h

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ i.c → k ∧ d.c.b → e ∧ a.f.g → h

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= d.o.i.k.m.a.c.h.b.e

SHRINKBODY

MQ(o.k.a.n.e)=true

No shrinking done.

ASSOCIATE

MQ(o.d.i.k.m.a.h.c.n.b.e)=true

MQ(o.d.i.k.m.a.h.c.b.e)=false

Associating with: l → n

BINARYSEARCH

MQ(o.k.h.e)=true

MQ(d.o.i.k.m.h.e)=true

MQ(d.o.i.k.m.a.h.e)=true

MQ(d.o.i.k.m.a.c.h.e)=true

Required addition b found.

MQ(d.o.i.k.h.b.e)=true

MQ(d.o.i.k.m.h.b.e)=false

Required addition m found.

MQ(d.o.k.m.h.b.e)=false

Required addition d found.

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ i.c → k ∧ d.c.b → e ∧ d.m.b → n ∧ a.f.g → h

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

EQ(h)= o.i.k.m.a.c.h.b.e

SHRINKBODY

MQ(o.k.h.c.n.b)=true

MQ(o.k.m.h.b.e)=false

Clause shrunk: m.b → n

Initial theory: a.c.b → e ∧ l → n ∧ f.g → h ∧ i.j → k ∧ c.b → o

Hypothesis: c.b → o ∧ i.c → k ∧ d.c.b → e ∧ m.b → n ∧ a.f.g → h

Target theory: d.c.b → e ∧ c.b → o ∧ i.c → k ∧ a.f.g → h ∧ m.b → n

---

Revision done using 10 EQs and 44 MQs.

Copyright ©Jignesh U. Doshi 2003.

Chapter 10

Conclusion

My work shows that revision of a subclass of Horn formulas requiring both additions and deletions can be done in time, that is logarithmic in the number of variables n (making the algorithm suitable in situations where n is big) and polynomial in the number of clauses m. I have implemented and tested this algorithm. The difficulty in the implementation was in generating counterexamples and query functions.

An extension to this work would be to enhance this algorithm to cover the entire set of Horn formulas that is, Horn formulas with non-unique and **'F'** heads, where head variables may also occur in bodies and may be revised.

Copyright ©Jignesh U. Doshi 2003.

Appendix A

Implementation Code

A.1 Clause.java

```
/*******************************************************
Created by Jignesh U. Doshi
Date: April 24, 2003
This file specifies the Horn clause data structure.
*******************************************************/
import java.util.*;
public class Clause {
    private HashSet body; // Clause body variable set.
    private String head;  // Clause head.
    public Clause(HashSet body, String head) { // Constructor.
        this.body=new HashSet(body);
        this.head=new String(head);
    }
    public Clause(Clause c) { // Constructor.
        this.body=new HashSet(c.getBody());
        this.head=new String(c.getHead());
    }
    public void print() {  // Prints clause in user readable form.
        Iterator i=body.iterator();
```

```
            while(i.hasNext()) {
                Global.window.process.appendText((String)i.next());
                if(i.hasNext()) Global.window.process.appendText(".");
            }
            Global.window.process.appendText("->"+head);
        }
        public String getHead() { // Returns head variable.
            return head;
        }
        public HashSet getBody() { // Returns body variable set.
            return body;
        }
        // Changes body set of a clause.
        public void setBody(HashSet newBody) {
            body.clear();
            body.addAll(newBody);
        }
}
```

A.2 Global.java

```
/*******************************************************
Created by Jignesh U. Doshi
Date: April 24, 2003
This file specifies the global variables, functions
and data structures that are used by other modules.
*******************************************************/
import java.util.*;
public class Clause {
    private HashSet body; // Clause body variable set.
    private String head;  // Clause head.
    public Clause(HashSet body, String head) { // Constructor.
        this.body=new HashSet(body);
        this.head=new String(head);
    }
    public Clause(Clause c) { // Constructor.
        this.body=new HashSet(c.getBody());
        this.head=new String(c.getHead());
    }
    public void print() {  // Prints clause in user readable form.
        Iterator i=body.iterator();
        while(i.hasNext()) {
            Global.window.process.appendText((String)i.next());
            if(i.hasNext()) Global.window.process.appendText(".");
        }
        Global.window.process.appendText("->"+head);
    }
    public String getHead() { // Returns head variable.
        return head;
    }
    public HashSet getBody() { // Returns body variable set.
        return body;
```

```
    }
    // Changes body set of a clause.
    public void setBody(HashSet newBody) {
        body.clear();
        body.addAll(newBody);
    }
}
```

A.3 RevUniExp.java

```
/*******************************************************
Created by Jignesh U. Doshi
Date: April 24, 2003
This file contains code for the GUI of the application
ReviseHorn.java
*******************************************************/
import java.awt.*;
import java.awt.event.*;
import java.util.*;
// GUI Window.
public class RevUniExp extends Frame implements ActionListener {
    public TextField initial;     // Input fields.
    public TextField target;
    public TextField univVar;
    public TextArea process;      // Output field.
    Label initialLabel;           // GUI components.
    Label targetLabel;
    Label hypLabel;
    Button revise;
    Button clear;
    Button help;
    Label varLabel;
    Label processLabel;
    Button algo;
    public RevUniExp() {          // Constructor
        // Layout manager object.
        RevUniExpLayout customLayout = new RevUniExpLayout();
        // Layout components added here.
        setFont(new Font("Helvetica", Font.PLAIN, 12));
        setLayout(customLayout);
        varLabel = new Label("Universe of Variables");
```

```
add(varLabel);
univVar = new TextField("");
add(univVar);
help = new Button("Help");
add(help);
help.addActionListener(this);
initialLabel = new Label("Initial Theory");
add(initialLabel);
initial = new TextField("");
add(initial);
targetLabel = new Label("Target Theory");
add(targetLabel);
target = new TextField("");
add(target);
processLabel = new Label("Revision Process");
add(processLabel);
revise = new Button("Revise Explanation");
add(revise);
revise.addActionListener(this);
clear = new Button("Clear");
add(clear);
clear.addActionListener(this);
algo = new Button("Algorithm");
add(algo);
algo.addActionListener(this);
process = new TextArea("");
process.setEditable(false);
process.setBackground(new Color(255,255,255));
add(process);
setSize(getPreferredSize());
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
```

```
                System.exit(0);
            }
        });
    }
    // All buttons' action listener.
    public void actionPerformed(ActionEvent e) {
      // for Revise Explanation button.
      if(e.getSource()==revise) {
        Global.init();
        Global.window.process.appendText("\n");
        // Make a set of all the variables.
        String input=Global.window.univVar.getText();
        StringTokenizer st = new StringTokenizer(input,".");
        while (st.hasMoreTokens()) {
          Global.univ.add(st.nextToken());
        } // Global.univ should not change after this line.
        // Get Initial theory and Target theory from the GUI and
        // validate them.
        if(ReviseHorn.getTheory(Global.INI) &&
           ReviseHorn.getTheory(Global.TAR)) {
            // Ask equivalence query to get a counterexample set.
            HashSet counterExample=Global.EQ();
            while(!counterExample.isEmpty()) {
                // Show the counterexample
                Global.window.process.appendText("EQ(h)=");
                Iterator i=counterExample.iterator();
                while (i.hasNext()) {
                    Global.window.process.appendText
                    ((String)i.next());
                    if (i.hasNext())
                      Global.window.process.appendText(".");
                    else Global.window.process.appendText("\n");
```

```
                }
                // Try shrinking clause body.
                if(!ReviseHorn.shrinkBody(counterExample)) {
                    // Shrinking failed, add new clause.
                    ReviseHorn.associate(counterExample);
                }
                Global.window.process.appendText
                ("----------------------------------------\n");
                // Print each step of revision.
                Global.printTheory(Global.INI);
                Global.printTheory(Global.HYP);
                Global.printTheory(Global.TAR);
                Global.window.process.appendText
                ("----------------------------------------\n");
                // Ask equivalence query again.
                counterExample=Global.EQ();
            }   // Print EQ and MQ counts.
            Global.window.process.appendText("Revision done
            using "+countEQ+" EQs and "+Global.countMQ+" MQs.\n");
        }
    }
    if(e.getSource()==clear) {
        Global.window.process.setText(""); // Clear all.
    }
    // Layout components added here.
    if(e.getSource()==algo) {
        try {
            // Open technical paper.
            Runtime rt = Runtime.getRuntime();
            Process p = rt.exec("cmd /c start
            reviseUniqueExp.pdf");
        } catch(Exception ex) {
```

```
            System.out.println(ex);
        }
    }
    if(e.getSource()==help) { // Show help in output box.
      Global.window.process.setText("");
      Global.window.process.appendText
      ("-------------------------------------\n");
      Global.window.process.appendText
      ("Using this application.\n");
      Global.window.process.appendText
      ("-------------------------------------\n");
      Global.window.process.appendText("Universe of variable
      should be entered as a.b.c.d.e.f.g (separated by '.')\n");
      Global.window.process.appendText("Initial theory should
      be entered as a.b->d ^ e.f->g
      (space only before and after '^')\n");
      Global.window.process.appendText("Target theory is also
      entered similarly as a.c->d ^ b.f->g\n");
      Global.window.process.appendText("Variables on the left
      of '->' are body variables and on the right is a head.\n
      Each clause is separated by ' ^ ' which
      represents a boolean AND.\n");
      Global.window.process.appendText("Both these theories
      should be unique explanations that is heads are,\n
      unique, not in any of the bodies, and intact i.e. heads
      are the same in initial and target theory clauses.\n");
      Global.window.process.appendText("\nOuput is the entire
      revision process shown step by step
      with each counterexample.\n Number of membership and
      equivalence queries asked in the revision process are
      also shown.\n");
      Global.window.process.appendText
```

```
            ("-----------------------------------------");
        }
    }
    public static void main(String args[]) {
        Global.window=new RevUniExp();
        Global.window.setResizable(false);
        Global.window.setTitle("Revising Unique Explanation
        - Jignesh Doshi and Judy Goldsmith");
        Global.window.pack();
        Global.window.show();
    }
}

// Component layout specified here.
class RevUniExpLayout implements LayoutManager {
    public RevUniExpLayout() {
    }
    public void addLayoutComponent(String name, Component comp) {
    }
    public void removeLayoutComponent(Component comp) {
    }
    public Dimension preferredLayoutSize(Container parent) {
        Dimension dim = new Dimension(0, 0);
        Insets insets = parent.getInsets();
        dim.width = 662 + insets.left + insets.right;
        dim.height = 385 + insets.top + insets.bottom;
        return dim;
    }
    public Dimension minimumLayoutSize(Container parent) {
        Dimension dim = new Dimension(0, 0);
        return dim;
    }
```

```
public void layoutContainer(Container parent) {
    Insets insets = parent.getInsets();
    Component c;
    c = parent.getComponent(0);
    if (c.isVisible())
      {c.setBounds(insets.left+8,insets.top+8,112,24);}
    c = parent.getComponent(1);
    if (c.isVisible())
      {c.setBounds(insets.left+128,insets.top+8,392,24);}
    c = parent.getComponent(2);
    if (c.isVisible())
      {c.setBounds(insets.left+528,insets.top+8,128,24);}
    c = parent.getComponent(3);
    if (c.isVisible())
      {c.setBounds(insets.left+8,insets.top+40,112,24);}
    c = parent.getComponent(4);
    if (c.isVisible())
      {c.setBounds(insets.left+128,insets.top+40,528,24);}
    c = parent.getComponent(5);
    if (c.isVisible())
      {c.setBounds(insets.left+8,insets.top+72,112,24);}
    c = parent.getComponent(6);
    if (c.isVisible())
      {c.setBounds(insets.left+128,insets.top+72,528,24);}
    c = parent.getComponent(7);
    if (c.isVisible())
      {c.setBounds(insets.left+8,insets.top+104,112,24);}
    c = parent.getComponent(8);
    if (c.isVisible())
      {c.setBounds(insets.left+256,insets.top+104,128,24);}
    c = parent.getComponent(9);
    if (c.isVisible())
```

```
            {c.setBounds(insets.left+392,insets.top+104,128,24);}
        c = parent.getComponent(10);
        if (c.isVisible())
            {c.setBounds(insets.left+528,insets.top+104,128,24);}
        c = parent.getComponent(11);
        if (c.isVisible())
            {c.setBounds(insets.left+8,insets.top+136,648,240);}
    }
}
```

A.4 ReviseHorn.java

```
/*******************************************************
Created by Jignesh U. Doshi
Date: April 24, 2003
This file contains code for the core of the revision
process. Routines for
  getting input theories,
  validating them,
  shrinking clauses,
  associating counterexamples and
  finding all the required body variable additions
are written in this file.
*******************************************************/
import java.util.*;
import java.io.*;
// Core class containing revision routines.
public class ReviseHorn {
    // Shrinks a hypothesis clause body if possible.
    static boolean shrinkBody(HashSet x) {
        Global.window.process.appendText("----------\n");
        Global.window.process.appendText("SHRINKBODY\n");
        Global.window.process.appendText("----------\n");
        Iterator i=Global.Hyp.iterator();
        while(i.hasNext()) { // For each hypothesis clause,
            Clause c=(Clause) i.next();
            HashSet temp=new HashSet(x);
            temp.retainAll(c.getBody());
            // check if it can be shrunk.
            if(temp.size()<c.getBody().size()) {
                HashSet check=new HashSet(temp);
                check.addAll(Global.headSet);
                check.remove(c.getHead());
```

```
                Global.window.process.appendText("MQ(");
                Global.printSet(check);
                Global.window.process.appendText(")="+
                Global.MQ(check, Global.TAR)+"\n");
                Global.countMQ--;
                if(!Global.MQ(check, Global.TAR)) {
                    c.setBody(temp);  // Shrinking done.
                    Global.window.process.appendText
                    ("Clause shrunk: ");
                    c.print();
                    Global.window.process.appendText("\n");
                    return true;
                }
            }
        }
        Global.window.process.appendText("No shrinking done.\n");
        // Could not shrink any hypothesis clause body.
        return false;
    }
    // Associate counterexample x to an initial theory clause.
    static void associate(HashSet x) {
        Global.window.process.appendText("----------\n");
        Global.window.process.appendText("ASSOCIATE\n");
        Global.window.process.appendText("----------\n");
        HashSet xdot=new HashSet(x);
        xdot.addAll(Global.headSet);
        HashSet d=new HashSet(xdot);
        d.removeAll(x);
        Iterator varSet;
        String input="n";
        String b="";
        Clause associated=new Clause(new HashSet(), "");
```

```
Global.window.process.appendText("MQ(");
Global.printSet(xdot);
Global.window.process.appendText(")="+
Global.MQ(xdot, Global.TAR)+"\n");
Global.countMQ--; // non-user membership query.
// While xdot is a true for Target theory,
while(Global.MQ(xdot, Global.TAR)) {
    Iterator i=d.iterator();
    while(i.hasNext()) {
        b=(String)i.next();
        // turn off a head not already in
        // hypothesis theory.
        if(!Global.hypHeadSet.contains(b)) {
            d.remove(b);
            break;
        }
    }
    xdot.remove(b);
    // Check if this makes it negative counterexample.
    Global.window.process.appendText("MQ(");
    Global.printSet(xdot);
    Global.window.process.appendText(")="+
    Global.MQ(xdot, Global.TAR)+"\n");
    Global.countMQ--;
} // Associate with first clause that is falsified.
Global.window.process.appendText("Associating with: ");
Iterator whichClause=Global.Ini.iterator();
while(whichClause.hasNext()) {
    associated=new Clause((Clause)whichClause.next());
    if(associated.getHead().equals(b)) {
        break;
    }
```

```
    }
    associated.print();
    Global.window.process.appendText("\n");
    BinarySearch(xdot, associated); // Call binary search.
}
// Find all additions required to associated clause.
static void BinarySearch(HashSet xdot, Clause associated) {
    Global.window.process.appendText("------------\n");
    Global.window.process.appendText("BINARYSEARCH\n");
    Global.window.process.appendText("------------\n");
    boolean firstIter=true;
    String input;
    Iterator varSet;
    HashSet temp=new HashSet(xdot);
    temp.retainAll(associated.getBody());
    HashSet xdotHeads=new HashSet(xdot);
    xdotHeads.retainAll(Global.headSet);
    temp.addAll(xdotHeads);
    // xdot symintersection body(associated).
    HashSet x=new HashSet(temp);
    HashSet start=new HashSet(temp);
    HashSet d=new HashSet();
    HashSet c=new HashSet();
    temp.clear();
    temp.addAll(xdot);
    temp.removeAll(x); // xdot intersection x.
    // While xdot and x are not the same,
    while(!temp.isEmpty()) {
        d.clear();
        d.addAll(temp);
        do {
            // If any more additions are needed.
```

```
        if(firstIter) {
            c.clear();
            firstIter=false;
        }
        else {// Turn off half the variables in c.
            c.clear();
            Iterator dVar=d.iterator();
            for(int count=0; count<d.size()/2; count++) {
                c.add(dVar.next());
            }
        }
        temp.clear();
        temp.addAll(x);
        temp.addAll(c);
        Global.window.process.appendText("MQ(");
        Global.printSet(temp);
        Global.window.process.appendText(")="+
        Global.MQ(temp, Global.TAR)+"\n");
        Global.countMQ--;
        // Check if any variable is left out.
        if(!Global.MQ(temp, Global.TAR)) {
            xdot.clear();
            xdot.addAll(temp);
            d.clear();
            d.addAll(c);
        }
        else {
            x.clear();
            x.addAll(temp);
            d.removeAll(c);
        }
    } while(d.size()>1);
```

```
            // Till we find a single required addition.
            if(!d.isEmpty()) { // Make the required addition.
                Iterator added=d.iterator();
                Global.window.process.appendText
                ("Required addition "+added.next()+" found.\n");
            }
            start.addAll(d);
            x.clear();
            x.addAll(start);
            temp.clear();
            temp.addAll(xdot);
            temp.removeAll(x);
        } // Try finding more required additions.
        // Remove head variables from the body.
        x.removeAll(Global.headSet);
        Clause newHypClause=new Clause(x,associated.getHead());
        // Add the final clause to the hypothesis.
        Global.Hyp.add(newHypClause);
    }
    // Validates input theory clauses.
    static boolean validate(int flag){
        HashSet bodySet=new HashSet();
        HashSet headSet=new HashSet();
        Iterator i;
        switch(flag) {
        case Global.INI: i=Global.Ini.iterator();
            break;
        case Global.TAR: i=Global.Tar.iterator();
            break;
        default:Global.window.process.appendText
            ("Bad arg to validate(). Exiting validate().\n");
            return false;
```

```
        }
        headSet.clear();
        bodySet.clear();
        while(i.hasNext()) {
            Clause c=(Clause)i.next();
            // Same heads in two or more clauses.
            if (headSet.add(c.getHead())==false) {
                Global.window.process.appendText
                ("Non unique Heads! Edit clause: ");
                c.print();
                Global.window.process.appendText("\n");
                return false;
            }
            bodySet.addAll(c.getBody());
        }
        HashSet temp=new HashSet(headSet);
        temp.retainAll(bodySet);
        // Head present in some body.
        if (!temp.isEmpty()) {
            Global.window.process.appendText
            ("Head in some body! Edit one of the clauses.\n");
            return false;
        }
        bodySet.addAll(headSet);
        // Variable not in the universe of variable.
        if(!Global.univ.containsAll(bodySet)) {
            Global.window.process.appendText
            ("Unknown variable! Edit one of the clauses.\n");
            return false;
        }
        return true;
    }
```

```
// Stores valid initial and target unique explanations.
public static boolean getTheory(int flag) {
    String input=new String();
    Iterator i;
    HashSet temp=new HashSet();
    switch(flag) {  // Get initial or final theory?
    case Global.INI:
        input=new String(Global.window.initial.getText());
        break;
    case Global.TAR:
        input=new String(Global.window.target.getText());
        break;
    default:
        Global.window.process.appendText
        ("Bad arg to getTheory()! Exiting getTheory().\n");
        return false;
    }
    StringTokenizer cl = new StringTokenizer(input," ^ ");
    while(cl.hasMoreTokens()) { // For each clause,
        String clause=cl.nextToken();
        StringTokenizer st = new StringTokenizer(clause,"->");
        // Check construct of the clause.
        HashSet body=new HashSet();
        if(!st.hasMoreTokens()) {
            Global.window.process.appendText
            ("Something wrong! Edit one of the clauses.\n");
            return false;
        }
        // Make a set of input body variables.
        StringTokenizer bd = new
        StringTokenizer(st.nextToken(),".");
        while (bd.hasMoreTokens()) {
```

```
            body.add(bd.nextToken());
        }
        // Check for the construct of an input clause.
        if(!st.hasMoreTokens()) {
            Global.window.process.appendText
            ("Something wrong! Edit one of the clauses.\n");
            return false;
        }
        String head=st.nextToken();
        // Make a clause object.
        Clause c=new Clause(body, head);
        switch(flag) { // Validate the clause.
        case Global.INI:Global.Ini.add(c);
            if (!validate(flag)) {
              Global.Ini.remove(c);
              return false;
            }
            break;
        case Global.TAR:Global.Tar.add(c);
            if (!validate(flag)) {
              Global.Tar.remove(c);
              return false;
            }
            break;
        }
    }
    switch(flag) {
    case Global.INI:i=Global.Ini.iterator();
        while(i.hasNext()) {
            Clause c=(Clause)i.next();
            // Make a set of heads.
            Global.headSet.add(c.getHead());
```

```
            }
            // Make a set of body variables.
            Global.bodySet.addAll(Global.univ);
            Global.bodySet.removeAll(Global.headSet);
            break;
        case Global.TAR: i=Global.Tar.iterator();
            temp.clear();
            while(i.hasNext()) {
                Clause c=(Clause)i.next();
                temp.add(c.getHead());
            }
            // Check if the target head set is the same
            // as initial theory head set.
            if(!temp.equals(Global.headSet)) {
                Global.window.process.appendText
                ("Target theory heads must be intact!
                Edit theory.\n");
                return false;
            }
            break;
        }
        // Print the theory in the user readable form.
        Global.printTheory(flag);
        // return success in getting valid theory.
        return true;
    }
}
```

Bibliography

[1] Dana Angluin. Learning regular sets from queries and counterexamples. *Inform. Comput.*, 75(2):87–106, 1987.

[2] Dana Angluin, Michael Frazier, and Leonard Pitt. Learning conjuctions of horn clauses. *Machine Learning*, 9:147–164, 1992.

[3] Jean-Paul Delahaye. *Formal Methods in Artificial Intelligence.* A Halsted Press Book, 1987.

[4] Jignesh Doshi and Judy Goldsmith. Revising unique explanation. *Proc. The 14th Midwest Artificial Intelligence and Cognitive Science Conference, MAICS*, pages 24–30, 2003.

[5] Allen Ginsberg, Sholom M. Weiss, and Peter Politakis. Automatic knowledge base refinement for classification systems. *Artificial Intelligence*, 35(2):197–226, 1988.

[6] Judy Goldsmith and Robert H. Sloan. More theory revision with queries. *Proc. ACM Symposium on Theory of Computing (STOC '00)*, pages 441–448, 2000.

[7] Judy Goldsmith, Robert H. Sloan, Balázs Szörényi, and György Turán. Theory revision with queries: Horn and other non-DNF forms, 2003. Submitted.

[8] Russell Greiner. The complexity of theory revision. *Proc. IJCAI*, pages 1162–1168, 1995.

[9] Moshe Koppel, Ronen Feldman, and Alberto Maria Segre. Bias-driven revision of logical domain theories. *Journal of Artificial Intelligence Research*, 1:159–208, 1994.

[10] L. Pitt M. Kearns, M. Li and L. Valiant. On the learnability of Boolean formulae. *Proc. 19th Annu. ACM Sympos. Theory Comput. (STOC '87)*, pages 285–294, 1987.

[11] Johann A. Makowsky. Why Horn formulas matter in computer science: Initial structures and generic examples. *Journal of Computer and System Sciences*, 34:266–292, 1987.

[12] Tom Mitchell. *Machine Learning.* McGraw Hill, 1997.

[13] R. J. Mooney. A preliminary PAC analysis of theory revision. *Computational Learning Theory and Natural Learning Systems.*, III:43–53, 1995.

[14] Chandra Reddy and Prasad Tadepalli. Learning horn definitions: Theory and an application to planning. *New Generation Computing*, 17(1):77–98, 1999.

[15] R. H. Sloan and G. Turan. On theory revision with queries. *Proc. 12th Annu. Conf. on Comput. Learning Theory.*, pages 41–52, 1999.

Vita

Date and Place of Birth: Godhra, India, April 9, 1980

Education: Bachelor of Engineering in Computer Engineering
University of Bombay, 2001

Professional Positions: Graduate Research Assistant
Department of Computer Science
University of Kentucky
Lexington, Kentucky, 2001–2003

Junior Systems Administrator
Math Sciences Computing Facility
University of Kentucky
Lexington, Kentucky, 2001

Web Developer
Freelance
Bombay, India, 1998–2001

Honors: Research Assistantship
University of Kentucky, 2001–2003

Professional Publications:

"Revising Unique Explanations" In *The 14th Midwest Artificial Intelligence and Cognitive Science Conference MAICS-2003*, University of Cincinnati, 2003.

Jignesh Doshi