Extending the Practicality of Theory Revision Systems through the Revision of Production System Rulebases

TECHNICAL REPORT 96-25

Patrick M. Murphy*

June 1996

*Correspondence should be directed via email to pmurphy@ics.uci.edu, or via post to UC-Irvine, Dept. of ICS, Irvine, CA 92717, USA.

UNIVERSITY OF CALIFORNIA

IRVINE

Extending the Practicality of Theory Revision Systems through the Revision of Production System Rulebases

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Patrick Michael Murphy

1996

Contents

List of Figures

List of Tables

Acknowledgments

I would first like to thank my advisor, Dennis Kibler, for taking me on as a student so late in the game, for letting me continue with the work that I had been doing, for giving me the encouragement, guidance and freedom to finish my dissertation, and for being an unofficial advisor during my early years as a graduate student. I would also like to thank Mike Pazzani, my previous advisor, and the other members of my doctoral committee, Nikil Dutt and Betty Olson, for their guidance.

During my six years as a graduate student, I have known many people whom I would like to acknowledge for their aid and guidance. Thanks should go to Kamal Ali, Cliff Brunk, Piew Datta, Caroline Ehrlich, Steve Hampson, Chris Merz, David Schulenburg, Eddie Schwalb, Doo Song and Jim Wogulis. Special thanks go to Jim Wogulis for aiding in the preparation of this dissertation and for making available the code for A3.

Lastly, heart felt thanks go to my wife Linda and children Sean and Shannon who gave me something to go home to at the end of the day and who provided me with the love, support, motivation and help to complete my dissertation.

Abstract of the Dissertation

Extending the Practicality of Theory Revision Systems through the Revision of Production System Rulebases

by
Patrick Michael Murphy
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1996
Professor Dennis F. Kibler, Chair

This dissertation addresses the problem of automatically revising production system rulebases using input-output mappings as the main source of information to guide the revision process. The main contribution of this work, the ability to revise production system rulebases, is important because production system rulebases are used so widely in industry. An implemented approach, CR2, is shown analytically and empirically to be able to revise these rulebases.

One important facet of this approach is an explicitly defined model, the revision problem space model, that was used to design CR2, and is used to understand and empirically analyze it. This model is important because, relatively speaking, the production system revision task is very difficult. The model allows for an understanding of when the revision system should succeed and fail.

Another facet of the approach is a set of techniques, the RIO techniques, that is used to identify revisions to the rulebase. These techniques were designed in the context of the revision problem space model. Unlike most Horn clause based revision systems, multiple techniques are needed to identify revisions independent of the problem with the rulebase and the information available to identify the problem.

In order to produce a revised rulebase that a domain expert would find comprehensible, a technique called rule structure filtering is used to avoid revisions that would produce "ill-structured" rules. This technique is shown analytically to produce more comprehensible revisions and is shown empirically to produce more accurate rulebases.

Chapter 1
Introduction

1.1 What is a Theory Revision System?

What is a theory revision system? Unfortunately, there is no universally agreed upon definition. In general, theory revision systems are systems that use an existing theory and a set of cases or constraints and background knowledge to produce a better theory. Beyond this definition, theory revision systems tend to belong to four, not completely disjoint, classes.

The first class consists of theory revision systems that don't actually revise the theory. They use the theory along with the cases to produce a more accurate theory than would be learned if only the cases had been used. These systems are often referred to as "knowledge-guide induction" systems because they use the extra knowledge available in the input theory to guide the induction from cases. Examples of knowledge-guided induction theory revision systems are ML-SMART (Bergadano & Giordana, 1988) and FOCL (Pazzani & Kibler, 1992).

The second class of theory revision systems more-or-less revise the input theory, but they do so after converting the theory to a new form. For example, KBANN (Towell, 1991) converts a propositional Horn clause theory to a neural net representation prior to revision. Another theory revision system of this class is RTLS (Ginsberg, 1988).

Members of the third class of theory revision systems are interactive. They revise or use the theory but require the aid of a user to guide the revision process. These systems are often referred to as knowledge acquisition systems. Examples of knowledge acquisition systems are TEIRESIAS (Davis, 1977), EXPERT (Weiss & Kulikowski, 1979), SEEK (Politakis & Weiss, 1984), MINERVA (Park & Wilkin, 1990), MOBAL (Morik, 1991), KR-FOCL (Pazzani & Silverstein, 1991) and $\mathcal{RLA}$ (Tangkitvanich & Shimura, 1992).

The last major class of theory revision systems consist of systems that revise the original theory directly. They follow the (sometimes explicit) bias of revising the theory using a minimal number of revisions. These systems are sometimes referred to as "true" theory revision systems. They, through some form of blame assignment, identify portions of the original theory that should be revised. Examples of "true" theory revision systems are SEEK2 (Ginsberg, Weiss & Politakis, 1985), EITHER (Ourston, 1991), NEITHER (Baffes & Mooney, 1993), FORTE (Richards, 1992), LATEX (Tangkitvanich & Shimura, 1993), CLIPS-R (Murphy & Pazzani, 1994), AUDREY (Wogulis, 1991), A3 (Wogulis, 1994) and CLARUS (Brunk & Pazzani, 1995).

The approach to theory revision, used in this research, is most closely related to existing "true" theory revision systems. Future discussions in this chapter will be based on this class of revision systems.

Beyond these four classes of theory revision systems, two additional characteristics that may be used to describe theory revision systems are the representation language of the theory and the form of cases and constraints that are used to guide revision. For most of the "true" theory revision systems, theories are usually represented using propositional or relation Horn clauses and the cases are facts that should and should not be provable by the theory.

For example, Table 1.1 shows a relation Horn clause theory that defines `grandfather/2` and `parent/2` in terms of `father/2` and `mother/2`. The clause `grandfather/2` declares incorrectly that person `X` is the grandfather of person `Z` if person `X` is the parent of some person `Y` and person `Y` is the parent of person `Z`. The two `parent/2` clauses declare that a person is the parent of another person if he/she is the father or mother of that other person.

Table 1.1 also shows cases that describe what should and should not be provable by the theory. Because `grandfather/2` is incorrectly defined, the negative case `not(grandfather(alice, sean))` is provable. Also shown is background knowledge that could be used by the revision system to revise the theory.

Based on the theory, cases and background knowledge shown in Table 1.1, the task of a theory revision system would be to identify that the addition of the literal `male(X)` to the `grandfather/2` clause would cause the incorrectly classified case `not(grandfather(alice, sean))` to be classified correctly. The correctly revised theory is shown below.

```
grandfather(X, Z) :- parent(X, Y), parent(Y, Z), male(X).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
```

Table 1.1. Example theory, cases and background knowledge.

Theory

```
grandfather(X, Z) :- parent(X, Y), parent(Y, Z).
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
```

Cases or Constraints:

```
grandfather(don, sean).
grandfather(don, shannon).
not(grandfather(don, patrick)).
not(grandfather(don, don)).
not(grandfather(alice, patrick)).
not(grandfather(alice, sean)).
```

Background Knowledge:

```
father(patrick, sean).
father(patrick, shannon).
father(don, patrick).

mother(linda, sean).
mother(linda, shannon).
mother(alice, patrick).

male(sean).
male(patrick).
male(don).

female(shannon).
female(linda).
female(alice).
```

1.2 The Failure of Current Research

Theory revision research has its greatest potential for practical benefit in the area of expert system knowledge base design and maintenance. Tens of thousands of man-hours could be saved each year by automating these two processes. Beyond the savings in time, theory revision research can also lead to more accurate knowledge bases by taking advantage of both the experience of the domain expert who helps to create the knowledge base and actual recorded case histories of the processes being modeled. True theory revision systems have the added potential of producing revised theories that tend to be close to the unrevised theories and are therefore more comprehensible to domain experts who create the original theories.

Unfortunately, current theory revision research has missed the mark by not concerning itself with the kinds of knowledge bases that are used most often by expert system designers. In particular, most research has explored the revision of Horn clause knowledge bases that perform classification tasks with backward chaining rules (Clancey, 1984). However, nearly all deployed knowledge-based systems make use of forward-chaining production rules with side effects. For example, two of the knowledge-based systems reported on at the 1993 Innovative Applications of Artificial Intelligence use CLIPS, a production system developed at NASA's Johnson Space Center. The remainder of the knowledge-based systems use ART, a commercial expert system that has many of the same features as CLIPS.

There are a variety of practical reasons why the production rule formalism is preferred to the Horn clause formalism in deployed expert systems. First, production rules are suitable for a variety of reasoning tasks such as planning, design and scheduling in addition to classification tasks that are addressed by logical rules. Second, most deployed knowledge-based systems must perform a variety of computational activities such as interacting with external databases or printing reports in addition to the "reasoning" tasks. The production system formalism allows such procedural tasks to be easily combined with the reasoning tasks. Third, production rule systems tend to be computationally more efficient. They allow the knowledge engineer to have more influence over the flow of control. Performance can be fine tuned. That is, in a Horn clause system, the rules indicate what inferences are valid. In a production system, the rules indicate both which inferences are valid and which inferences should be made at a particular point.

1.3 Objectives of this Dissertation

The main objective of this dissertation is an understanding of issues related to the revision of production system rulebases. This dissertation will describe and analyze, both empirically and analytically, an implemented approach called CR2. Empirical results are based on a set of artificial and natural rulebases. All rulebases dealt with in this dissertation are represented using a subset of the CLIPS (Giarratano & Riley, 1993) production system language.

One issue addressed in this research concerns the type of information used to constrain the revision process. Final fact-list constraints, which are constraints on the contents of the fact-list after execution of an instance, are the main source of constraining information. These constraints are somewhat analogous to the examples used by most Horn clause based revision systems. Weaknesses in the ability to accurately revise a production system rulebase using only final fact-list constraints are evaluated and described.

As the task of identifying revisions to production system rulebases tends to be more difficult than the analogous task for Horn clause theories, a model of the revision problem space is introduced and described. This model is used to demonstrate which classes of revision problems are easy and difficult to handle.

In order to cover most partitions of the revision problem space, a set of revision identification and ordering techniques was designed using insights gained from the revision problem space model. These techniques are studied and analytically and empirically shown to be useful at revising production system rulebases.

Consistent with the practical bent of revising production system rulebases, the number of full revision evaluations made during each hill-climbing step was designed to be efficiently limitable. The effect of using such a limit is empirically analyzed.

Since most theory revision research has not explicitly identified and described analogous models, the identification and use of a revision problem space model is a significant contribution of this research.

Another issue addressed in this research concerns the identification of revisions that would make sense to a human reviser. Most theory revision approaches concentrate on efficiently using the examples to identify highly evaluated revisions to make to the theory. They ignore the problem of trying to determine which of the highly evaluated revisions would make the most sense to a human reviser. To this end, an approach taken by CR2 includes an extension of the idea of rule models, originally demonstrated in TEIRESIAS (Davis, 1977). This approach is fully automated and filters revisions that would produce "ill-structured" rules. Analytical results are presented that show why this approach should work to produce more understandable revisions and empirical results are presented that show when it works to improve performance.

1.4 Overview of this Dissertation

This dissertation describes the task of revising production system languages, an approach to the problem, an implemented system, and an evaluation of the approach. The following summarizes the contents of this dissertation.

- **Chapter 2: Background**

  Presents background information related to the task of theory revision and productions system rulebases. Describes the forward chaining production system representation language CLIPS. Also describes constraint information and biases that may be used to revise production system rulebases.

- **Chapter 3: Revision Problem Space**

  Presents a model for understanding the revision problem space. Defines five characteristics, i.e. problem type, problem class, rule firing sequence deviation class, problem location and constraint violation location, which will be used to partition the revision problem space. An understanding of the revision problem space model is necessary for an understanding of Chapters 4, 5 and 6.

- **Chapter 4: CR2: A Production System Rulebase Reviser**

  Presents a description of CR2, an implemented system that is capable of revising production system rulebases. Describes SPR, the main component of CR2. Describes the RIO techniques that are used to identify and order high-level revisions. Shows how revisions are expanded, evaluated and filtered using a rule structure similarity metric.

- **Chapter 5: Singleton Problem Reviser: Results & Evaluation**

  Presents the results of experiments based on the use of SPR at revising singleton revision problems. Empirically shows strengths and weaknesses of the revision problem space model introduced in Chapter 3. Presents ablation studies that demonstrate the need for each of the RIO techniques.

- **Chapter 6: CR2: Results & Evaluation**

  Presents the results of experiments based on the use of CR2 at revising rulebases with multiple problems. Demonstrates that CR2, using SPR, is capable of revising rulebases with multiple problems even though such rulebases violate the assumptions under which SPR was developed.

- **Chapter 7: Rule Structure Filtering**

  Presents a description of and reasons for rule structure filtering. Shows when rule structure filtering is and is not an appropriate bias.

- **Chapter 8: A Comparison to Other Approaches**

  Presents a comparison of CR2 with four other theory revision systems. Presents an analytical and empirical comparison between CR2 and A3 and an analytical-only comparison with the other systems.

- **Chapter 9: Conclusions**

  Reviews the over-all contributions of this dissertation, describes limitations and provides directions for future research.

Chapter 2
Background

2.1 Chapter Overview

This chapter presents a background of some general issues related to theory revision. Discussion begins with a basic overview of theory representation languages including the specific production system language used in this research. Later sections cover theory revision constraints and preferences, evaluation metrics and revision selection biases. Throughout this chapter, specific details of this research are presented in the context of other existing and potential research.

2.2 Theory Representation Languages

Most theory revision systems revise theories that are represented using Horn clauses, e.g. EITHER, FORTE, A3 and CLARUS. The earliest of theses systems revised proposition Horn clause theories, e.g. EITHER. Recent theory revision systems are able to revise relational Horn clause theories, e.g. FORTE, A3 and CLARUS. The propositional Horn clause representation language is a proper subset of the more expressive relational Horn clause language. Table 1.1 shows an example of a relational Horn clause theory, and Table 2.1 shows an example of a propositional Horn clause theory.

The main direction in theory revision research has been to construct theory revision systems that can handle more expressive theory representation languages. For example, EITHER, which is only able to revise propositional Horn clauses, was designed first. Next, FORTE was designed to revise negation-free relational Horn clause theories. Lastly, A3 was designed to revise relational Horn clause theories with negation.

Table 2.1. Example propositional Horn clause theory.

```
cup          :- liftable, stable, open_vessel.
liftable     :- light, has_handle.
stable       :- has_flat_bottom.
open_vessel  :- has_cavity, concave_up.
```

In this research, the direction taken has been to understand the issues involved in revising production system rulebases. While the production system language is not necessarily more expressive than the relational Horn clause language, it has characteristics that has made it a more popular implementation language for expert systems knowledge bases.

The specific production system language dealt with in this research is a subset of the language used by CLIPS. The CLIPS production system language was chosen because it is a very popular expert system implementation language and is closely related to the language used by ART and OPS5 (Brownston, 1985), other very popular production system languages.

The CLIPS language is similar to a variety of production system languages, in that it consists of a rulebase, an agenda (an ordered sequence of rule activations), and a fact-list (working memory). An activation consists of a rule and a set of bindings for variables in the rule (see Figure 2.1).

Rule execution proceeds as follows. While there are activations on the agenda, the top-most activation is removed and executed. If the execution of the activation alters the fact-list by asserting new facts or retracting existing facts, activations from newly satisfied rules are added to the agenda, and existing rule activations that are no longer satisfied are removed from the agenda (see Table 2.2).

The position within the agenda where a new rule activation is placed is a function of the rule's salience and the current conflict resolution strategy. For this research the *depth strategy* is used. With the depth strategy, an activation is placed above all activations of equal or lower salience and below all activation of higher salience. The results of this research are also applicable to the *breadth strategy*, where an activation is placed above all activation of lower salience and below all activations of equal or higher salience.

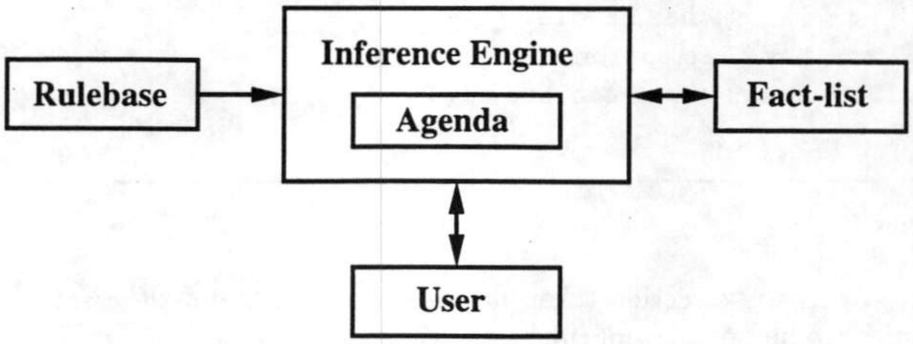


Figure 2.1. Main components of a CLIPS production system.

Table 2.2. CLIPS production system inference engine.

```
Execute_Rulebase(agenda, rulebase, fact-list)
  {
    while agenda is not empty
      activation = top_activation(agenda)
      rule = activation_rule(activation)
      bnds = activation_bindings(activation)
      for each action in rule
        if action is an assert
          fact-list = Assert_Fact(action, bnds, fact-list)
        elsif action is a retract
          fact-list = Retract_Fact(action, bnds, fact-list)
        else /* all other actions */
          bnds = Execute_Other_Action(action, bnds)
      agenda = Update_Agenda(agenda, rulebase, fact-list)
    return fact-list
  }
```

Initialization of the agenda occurs when the rulebase is first loaded and when initial facts are manually asserted to the fact-list. The initial load of the rulebase causes the agenda to be loaded with activations formed from rules that are satisfied by an empty fact-list. Each assertion of an initial fact may produce new activations or may cause the deletion of existing activations.

Table 2.3 shows some examples of CLIPS rules. A rule has an optional initial declaration section, a left-hand side (LHS) or antecedent of the rule, and a right-hand side (RHS) or consequent of the rule. The RHS follows the `=>` symbol. The rule in Table 2.3c has a declaration section which declares that the rule's salience is 10 (when omitted, the salience of a rule defaults to 0).

The LHS consists of conditional elements (CEs) that each must be satisfied to form a rule activation. The two types of CEs dealt with in this research are the pattern CE and the test CE. Pattern CEs are satisfied by matching a fact in the fact-list. They are represented as a sequence of constants and variables. Test CEs correspond to variable-variable or variable-constant comparisons. CEs may be negated. A negative CE is satisfied if its pattern or test is unsatisfied.

The RHS of a rule consists of an ordered sequence of actions. The two most common actions, assert and retract, modify the fact-list. The assert action adds a ground fact to the fact-list. The retract action removes a fact from the fact-list. The retract action in the rule shown in Table 2.3d retracts the fact that satisfies the CE `(female-head 2)`.

Another action, the bind action, binds the return value of a function to a variable newly introduced in the RHS. In Table 2.3b, the result of calling the function `ask-question()` with parameter `shape-tail?`, is stored in the variable `?response`. The function `ask-question()` prints the question to the screen and waits for a response. The only other action, the printout action, prints information to the user terminal.

2.3 Revision Constraints & Preferences

A necessary input to any theory revision system is a set of rulebase specific constraints and preferences that are used by the system to help determine which revisions to make to the theory. These constraints describe how the system should perform. For Horn clause theory revision systems, these constraints have usually been in the form of instances that describe what should and should not be provable by the revised theory. Other constraints in the form of, for example, variable types,

Table 2.3. Examples of CLIPS rules.

A. Rule from Student Loan Rulebase

```
(defrule continuously-enrolled
   (never-left-school)
   (enrolled ?School ?Units)
   (school ?School)
   (test (> ?Units 5))
   =>
   (assert (continuously-enrolled)))
```

B. Rule from Nematode Rulebase

```
(defrule Aphelenchoidea-tail-shape
   (Superfamily Aphelenchoidea)
   (not (tail-shape ?size))
   =>
   (bind ?response (ask-question shape-tail?))
   (assert (tail-shape ?response)))
```

C. Rule from Auto Diagnosis Rulebase

```
(defrule determine-battery-state2
   (declare (salience 10))
   (charge-state battery dead)
   =>
   (assert (repair Charge-the-battery)))
```

D. Rule from Nematode Rulebase

```
(defrule head-shape-2
   ?head <- (female-head 2)
   =>
   (retract ?head)
   (assert (female-head box-grid)))
```

have been used to constrain and preference revision selection. A major area of expansion in theory revision research, as well as in all of machine learning, has been to increase the ability of these systems to use new forms of constraints and preferences.

For theory revision systems that revise production systems, many forms of constraints and preferences are possible. The constraints, most analogous to the constraints used by Horn clause revision systems, are constraints on the contents of the final fact-list. Because the order of rule execution may be significant during the execution of production system rulebases, constraints and preferences on the order of rule and action executions are also possible.

Another important form of constraints are constraints that take into consideration the structure of production systems rulebases. For example, an analysis of many CLIPS rulebases has shown that the rule groups tend to exist. Rules in the same group tend to have a similar structure and/or a particular purpose.

For example, in the nematode identification rulebase, rule execution proceeds through a series of these rule groups. Figure 2.2 shows the potential flow of rule execution through the groups. The purpose of the rule group *classify* is two fold. It produces a classification that is modified by later rules and added to a database of previous classifications. It also produces an explanation of the classification that is printed by rules in the rule group *print classification*.

Some useful constraints that are possible in the context of rule groups are constraints on the contents of the fact-lists at the time that rule execution leaves a rule group. For example, upon exiting the rule group *classify*, fact-list constraints could be used to make sure that the correct classification and explanation facts were asserted.

The types of constraints used in this research include constraints on the final fact-list, constraints on the maximum number of rule executions for each instance, variable typing, constraints on the types of facts that may be assertable by a rule, and constraints on the types of revisions that may be used to revise the rulebase.

Final fact-list constraints and rule execution limit constraints are soft constraints and are used to evaluate revisions to the rulebase. They are described in the following sections. The other constraints are hard constraints. They are used to determine which types of revisions are legal. See Section 4.4 for a more detailed description of these constraints.

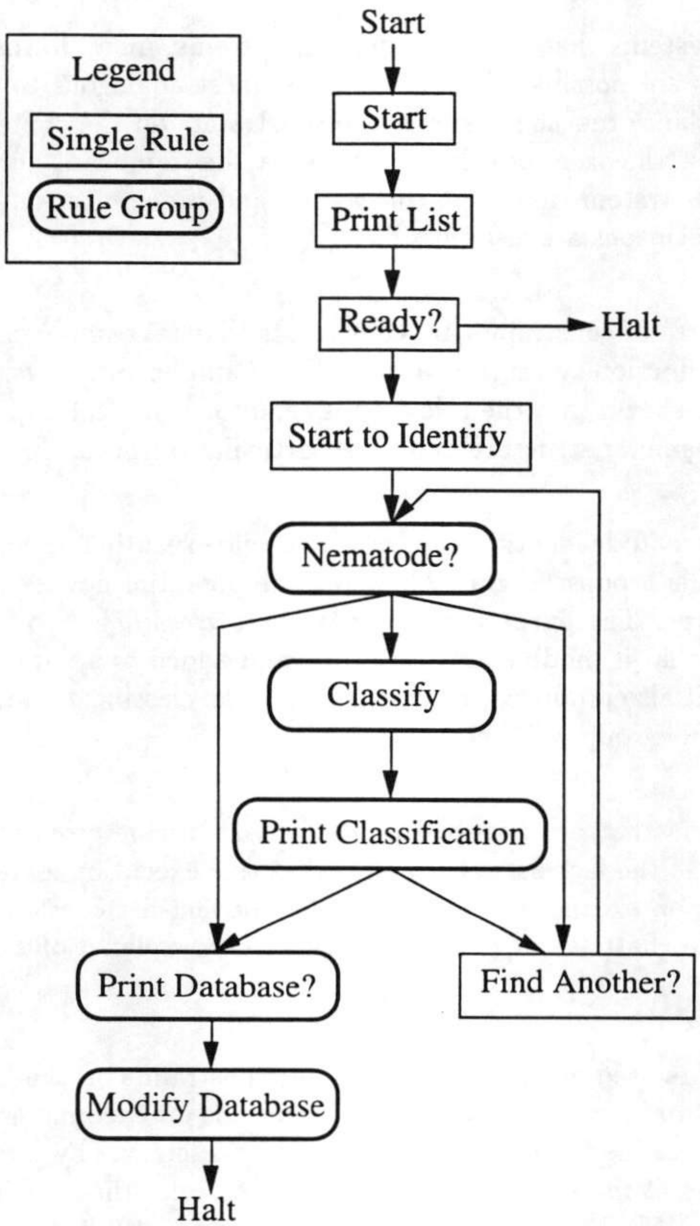


Figure 2.2: Rule groupings and flow-of-control for nematode identification rulebase.

Table 2.4. Example instance from auto diagnosis rulebase.

```
Initial State Information:
 Initial Facts:  NIL
 Function Call Bindings:
  (ask-question what-is-surface-state-of-the-pnts?)  → normal
  (yes-no-p does-the-engine-start?)  → yes
  (yes-no-p does-the-engine-run-normally?)  → no
  (yes-no-p does-the-engine-rotate?)  → no
  (yes-no-p is-the-engine-sluggish?)  → no
  (yes-no-p does-the-engine-misfire?)  → no
  (yes-no-p does-the-engine-knock?)  → no
  (yes-no-p is-the-output-of-the-engine-low?)  → no
  (yes-no-p does-the-tank-have-any-gas-in-it?)  → yes
  (yes-no-p is-the-battery-charged?)  → yes
  (yes-no-p is-conductivity-test-for-ign-coil-pos?)  → no
Constraints on Execution of Rulebase:
 Final Fact-list Constraints:
  Target:  (repair timing-adjustment)
  Others:  NIL
```

2.3.1 Final Fact-List Constraints

An *instance* has two components, initial state information and constraints on the final fact-list. Initial state information consists of a set of initial facts to be loaded into the fact-list before execution of the rulebase, and a set of bindings that relates a function call (represented by a function name and values for its arguments) to its return value. For example, the function `ask-question()` with argument `What-is-surface-state-of-pnts?` may return burned for one instance and contaminated for another instance. Constraints on the final fact-list are divided into two types, *target concept* constraints and other constraints (*intermediate concepts*). Other constraints tend to refer to subconcepts induced while forming the target concept for a classification system.

Final fact-list constraints are a set of CLIPS pattern conditional elements that should be satisfied by the facts of the final fact-list. For example, for the constraint `(repair Add-gas)` to be satisfied, it should match some fact in the final fact-list. On the other hand, the constraint `(not (working-state engine` `normal))` is satisfied if it does not match any fact in the final fact-list. Table 2.4 shows an example training instance for the auto diagnosis rulebase. Final fact-list constraints need not be ground. Most existing systems are only able to use ground constraints.

2.3.2 Rule Execution Limit Constraints

Rule activation execution limit constraints are constraints on the maximum number of activations that may be executed during the evaluation of an instances. Each instance may be associated with a different limit (elimit). In this research, all instances associated with a specific rulebase have the same elimit.

The purpose of an elimit constraint is to avoid infinite loops that can occur during the evaluation of instances. The determination of what elimit to associate with a domain depends on an understanding of the domain. For domains where no rule should execute more than once during the execution of an instance, the elimit can be set to the number of rules in the rulebases. Elimit constraints are also new to theory revision research.

2.4 Revision Evaluation

The metric typically used by Horn clause based revision systems to guide the selection of revisions is some form of error (or accuracy). Error has typically been measured in terms of the number of misclassified instances.

In this research, a lexicographic evaluation metric is used to identify the best revisions during evaluation over the instances. The primary evaluation key is the number of elimit constraint violations. The secondary key is the number of violated final fact-list constraints. Avoidance of infinite loops was deemed to be more important than avoidance of final fact-list constraint violations.

In later chapters, evaluation of the quality of the revised rulebase will also be measured in terms of both error rate and the number of elimit constraint violations. Most of the focus will be placed on error rate.

2.5 Revision Biases

Revision biases are, for the most part, rulebase independent constraints and preferences that may be used to guide the selection of revisions. Two forms of revision biases that have been used in existing work include the minimum revision distance bias used by A3 and linguistic-based semantics used by CLARUS.

The revision distance bias used by A3 is used to select among equally evaluated revisions. When multiple equally evaluated revisions exist, the revision that modifies the theory least, according to a literal-level distance metric, is selected to permanently revise the theory.

CLARUS's linguistic-based semantics technique is a preference bias that uses lexical cohesiveness to order revisions. In other words, revisions that generate more lexically cohesive rules are preferred over revisions that generate less lexically cohesive rules. Lexical cohesiveness is determined by mapping terms in a rule to elements in a lexical hierarchy and then measuring the distance between elements in the hierarchy.

A revision bias used in this research, rule structure filtering, takes advantage of the rule structure to identify rule groups. Rule structure filtering is used as a final selection measure after evaluation. For each of the best evaluated revisions, the rule being revised by the revision is associated with the best rule group consistent with the rule. After revision, the best consistent group is again identified. If the quality of the best group after revision is worse than the quality of the best group before revision, the revision is removed. The quality of a rule group depends on the similarity between rules in the group and the number of rules in the group. Chapters 4 and 7 present more detailed descriptions of rule structure filtering.

2.6 Chapter Summary

Most existing theory revision research has been done using Horn clause represented theories. Unfortunately, in practice, theories represented using production systems rulebases tend to be used much more often than Horn clause based theories. This research extends the utility of existing theory revision research by identifying an approach that is able to revise CLIPS rulebases (a very popular production system language).

In the context of production system theory revision research, many forms of constraints and preferences may be used to guide the revision process. For this research, constraints on the final fact-list and on the maximum number of rule executions per instance are used to form a lexicographic evaluation metric that rates revisions. Other constraints that are used to narrow the revision search space are constraints on variable typing, constraints on the types of facts that may be asserted by a rule, and constraints on the types of revisions that may be used to revise a rule. The use of final fact-list constraints and elimit constraints is new to theory revision research.

Many theory revision systems use some form of explicit preference bias for selecting among revisions. In this research, a new bias for revisions that do not produce ill-structured rules is used. This bias takes into account the original structure of the rulebase.

Chapter 3
Revision Problem Space

3.1 Chapter Overview

Most theory revision systems use some form of blame-assignment technique to identify likely high-level revisions. The high-level revisions are expanded and evaluated to identify the best low-level revision to make to the theory.

In order to identify the set of likely high level revisions, researchers have used various approaches. Some approaches are homogeneous, for example in A3 (Wogulis, 1994), a single assumption-based technique is used to suggest a subset of likely high-level revisions. In other systems like EITHER (Ourston, 1991), depending on a characteristic of the revision problem (e.g., the concept is overly general or overly specific), a different technique is used.

For the approach described in this research, multiple techniques were developed for identifying likely revisions (see Chapter 4). These techniques are designed to identify revisions in different partitions of the *revision problem space*.

In this chapter, the concept of the revision problem space is described as well as a model for characterizing and understanding revision problems in the context of revising production system rulebases. The main goal of formally modeling this space is to enable an understanding of the strengths and weaknesses of some of the approaches used in this research. The importance of this model will be understood during the empirical analyses performed in Chapter 5.

3.2 Revision Problem Space

The revision problem space is the space of all revision problems. A revision problem, in general, is the task faced by a revision system that is attempting to revise a rulebase in the context of constraints and preferences. The components of

Table 3.1. Singleton problem types.

- a rule with a missing or extra action
- a rule with a missing or extra CE
- a rule with too high or too low salience
- an extra rule

a revision problem may include, a problematic rulebase, constraining information that may be used to locate problems in the rulebase, symptoms of the problems and an understanding (or model) of rule execution.

The revision problem space may be looked at as a set of cases or situations that happen during the process of solving revision problems. For example, in the context of playing a hand of poker. The "poker problem space" would consist of those situations that a poker player may find himself in, e.g. what cards does he hold, how much money does he have to bet with, does he believe that other players are cheating, what are the rules of the specific games being played, etc. Every possible set of answers to these questions corresponds to a different element of the poker problem space.

The revision problem space modeled in this research is over rulebases that have only a single problem (Table 3.1). The rule with the problem is defined as the *problem rule*. Note, the approach presented in this research is able to handle rulebases that have multiple problems.

The characteristics that are used to describe this space are the problem type and class, the deviation of the rule firing sequence from the correct rule firing sequence, the general location of problems in rule firing sequences and the location of constraint violations (CVs) in rule firing sequences. For this research, in the context of a problematic rulebase, one or more revision problems are associated with each instance/CV pair. Note, in the remainder of this dissertation, CV will be used as an abbreviation for constraints violation.

3.3 Problem Class

The *problem class* of a rulebase is directly related to the problem type of the rulebase. For reasons that will become clear later, the possible types of problems, missing or extra action, missing or extra CE, too high or too low salience, and

$$\begin{tikzcd}
{} \arrow[r, "A"] & {} \arrow[r, "B"] & {} \arrow[r, "C"] & {} \arrow[r, "D"] & {} \arrow[r, "E"] & {}
\end{tikzcd}$$

Figure 3.1. Correct rule firing sequence.

extra rule, are partitioned into three classes. The problem types missing and extra action are grouped together to form the problem class *wrong action*. The problem types missing CE, high salience and extra rule form the problem class *under-constrained rule*. The remaining two problem types, extra CE and low salience form the problem class *overly-constrained rule*.

3.4 Rule Firing Sequence Deviation Class

The *rule firings sequence deviation class* is defined in the context of an instance and an errorful rulebase. Given a correct rulebase and an instance, the sequence of rules executed by the rulebase may look like the sequence of rule firings shown in Figure 3.1. When an error is added to the rulebase however, and the instance is re-executed using the errorful rulebase, the sequence of rule firings may look different and will look similar to one of the sequences shown in Figure 3.2. Note, sequences of fired rule activations, not rules, are actually used to determine the rule firing sequence deviation class.

Each of the sequences shown in Figure 3.2 shows a different class of deviation from the correct rule firing sequence (Figure 3.1). The set of deviations is exhaustive. Next to each sequence in Figure 3.2 is a label which is used to identify the deviation class.

The sequence labeled *correct* should be associated with all rule firing sequences that do not deviate from the correct rule firing sequence. Note, this does not imply that there are no CVs associated with this class of rule firing sequences. One of the rules in the sequence could have executed a wrong action that doesn't change the rule firing sequence but could cause a CV.

The sequence labeled *missing* is correct up to the execution of rule B but then halts and is missing the execution of rules C, D and E. This class of deviations refers to all rule firing sequences that are identical to the correct sequence up to a certain point and then halt prematurely. This class includes the null sequence of rule firings (no rules fire).

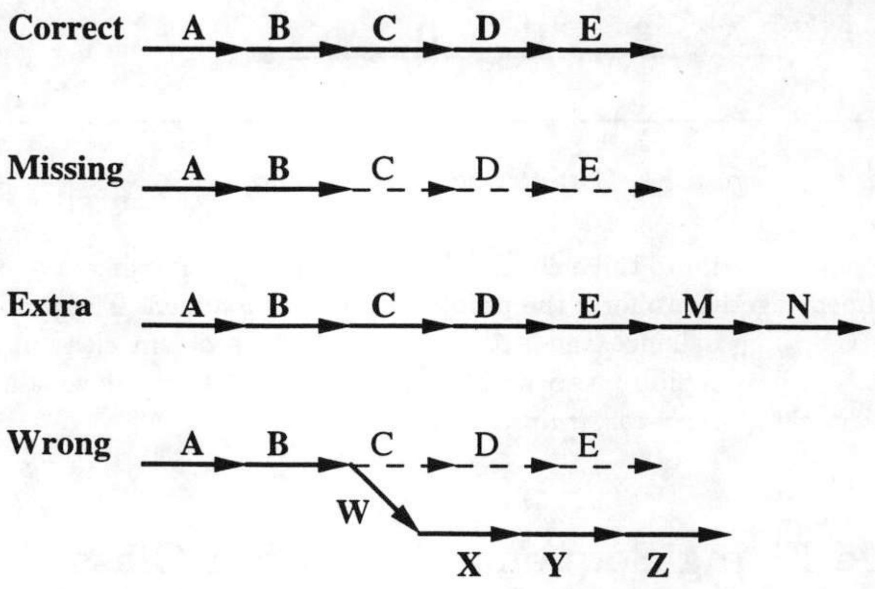


Figure 3.2. Rule firing sequence deviation classes.

The sequence labeled *extra* is the class of deviations that execute the correct sequence of rules followed by one or more additional rules, e.g. M and N.

The final sequence, labeled *wrong*, is the class of rule firing sequence deviations that execute correctly up to a certain rule and then deviate by executing one or more rules not in the correct sequence. This class of rule firing sequences also includes those sequences that start out by executing a rule different from the correct first rule.

3.5 Problem Locations

Problem location is defined in the context of an instance and a rulebase, and is constrained by the rule firing sequence deviation class of the instance. All errorful instances, instances that produce one or more CVs, have problem locations. The problem location is associated with the firing of the problem rule. Only errorful instances have problem locations.

The potential problem locations for an instance depend on its rule firing sequence deviation class. Figure 3.3 defines these locations as a function of deviation

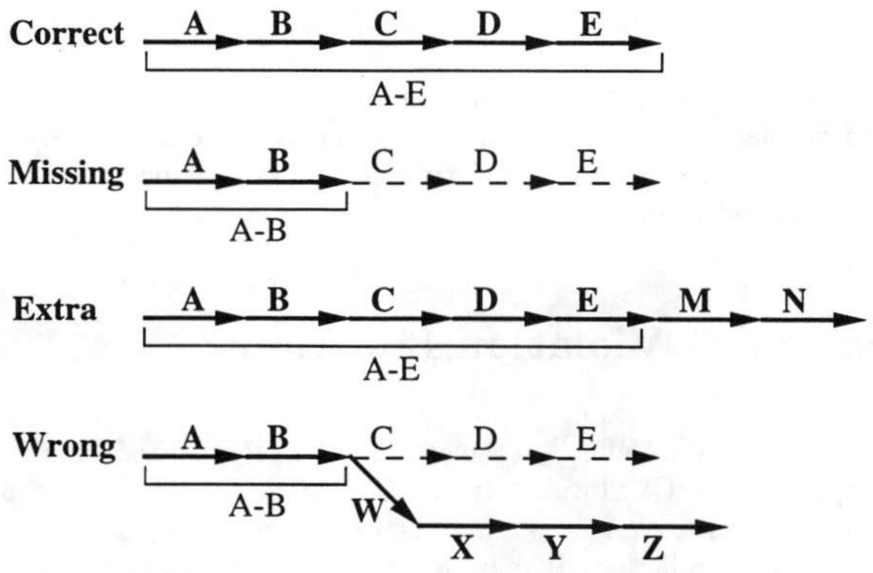


Figure 3.3. Problem location as a function of rule firing sequence deviation class.

class. Problem class is constrained in the context of deviation class and problem location.

If the deviation class is correct, then the potential problem locations are the entire sequence of rule firings, i.e. subsequence A-E. The problem class is constrained to be wrong action.

If the deviation class is *missing*, then the problem locations are constrained to include, rules A, B and C. A problem location in the subsequence A-B (rules A and B) would be constrained to have a problem class of *wrong action.* If rule C was the problem location, it would be constrained to have an problem class of *overly-constrained rule* (specifically, extra CE).

If the deviation class is *extra*, then the problem locations are constrained to be subsequence A-E or rule M. If the problem location was in subsequence A-E, the problem class would have to be *wrong action.* If the problem location was rule M, then the problem class would be *under-constrained rule* (specifically, missing CE).

If the rule firing sequence deviation class is *wrong*, then the problem location would be one of three locations: either subsequence A-B, rule C or rule W. If the problem location was subsequence A-B, then the problem class would have to be *wrong action.* If the problem location was rule C, then the problem class would have to be *overly-constrained rule.* Finally, if the problem location was rule W, then the problem class would be *under-constrained rule.*

Independent of deviation class, potential problem locations tend to be highly correlated with problem class. Subsequences A-E and A-B are always associated with the problem class *wrong action* and rule C is always associated with the problem class *overly-constrained rule.*

3.6 Constraint Violation Locations

The location of a CV is defined in the context of a rulebase and the instance that is associated with the CV. Intuitively, the CV location is the direct cause of the CV. For example, an extra fact CV may be caused by the assert action that inappropriately added the fact, or it may be caused by a missing retract action that should have retracted the fact. The CV location can only be determined in the context of the correct rule firing and action execution sequence. It is important to understand that the location of a CV can be a fired or unfired rule (executed or unexecuted action).

Each CV may be associated with multiple CV locations. For example, a positive final fact-list CV may have a different CV location for each missing fact that may have satisfied the constraint. A negative final fact-list CV may have a CV location associated with each fact that incorrectly matched the constraint.

Unlike the problem class and problem location, CV location is not constrained by the rule firing sequence deviation class, though there are useful subsets of the potential CV locations that are defined in terms of deviation class. Figure 3.4 shows these regions, i.e. subsequences A-E, A-B, C-E, M-N and W-Z. Note, for the deviation classes correct and extra, the CV locations are associated with rules that fired, but for deviation classes, missing and wrong, some CV locations, i.e. subsequence C-E, are associated with rules that did not fire, but should have fired.

3.7 Chapter Summary

The main goal of this chapter was to describe the concept of the revision problem space and a taxonomy that characterizes the revision problem space used in this research. Elements of the taxonomy include, problem type and class, rule firing sequence deviation class, problem location and CV location. Where applicable, constraints between these characteristics have also been presented.

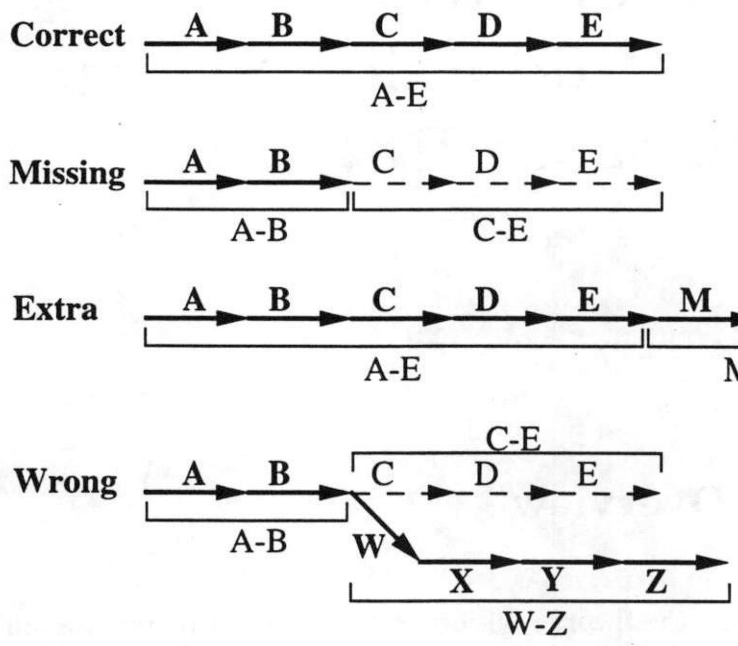


Figure 3.4. Useful constraint violation locations.

The purpose in formally describing the revision problem space model is to allow an understanding of the conditions under which the approaches used in this research will be able to revise production system rulebases. An understanding of the contents of this chapter is important as background for understanding Chapter 4 (a system description) and the empirical validations in Chapters 5 and 6.

Chapter 4
CR2: A Production System Rulebase Reviser

4.1 Chapter Overview

In previous chapters, the theory revision problem, theory representation languages, and an ontology for modeling the theory revision problem space associated with the revision of production system rulebases were discussed. This chapter describes CR2, a revision system for the revision of CLIPS-like production system rulebases.

CR2 is described first, followed by a description of SPR, the main component of CR2. Later sections include a description of the RIO techniques, high-level revision expansion and evaluation, rule structure filtering and available forms of constraints.

4.2 CR2

CR2 is organized as a hill-climbing system which takes as input a rulebase and set of instances and returns a revised rulebase. Multiple revisions to the rulebase are accomplished through a series of single revision hill-climbing steps. Table 4.1 shows a high-level description of CR2.

At the heart of CR2 is SPR, a singleton-problem reviser. SPR is designed to revise rulebases that have only a single problem, e.g. a missing or extra CE, a missing or extra assert or retract action, too high or low salience or extra rule. Within the context of CR2, SPR may be used to revise rulebases with multiple problems.

Table 4.1. High-level description of CR2.

```
CR2(rulebase, instances)
   {
     while evaluation(rulebase, instances) is improving
       rulebase = SPR(rulebase, instances)
     return rulebase
   }
```

4.3 SPR

SPR is the main component of CR2. It was designed in the context of the revision problem space model described in Chapter 3. SPR can repair problems associated with most areas of the revision problem space. The strengths and limitations of SPR are empirically demonstrated in Chapter 5.

SPR is organized as an operator-based rulebase optimization system. A set of revisions is first identified from the set of possible singleton revisions using a set of revision identification of ordering (RIO) techniques, then filtered using evaluation over the instances and then further filtered using a rule structure similarity metric. From the remaining revisions, a revision is randomly selected and used to revise the rulebase. Table 4.2 shows a high-level description of SPR.

4.3.1 RIO Techniques

The RIO techniques form the component of SPR that identify the initial set of revisions. Each RIO technique is designed to identify revisions associated with

Table 4.2. High-level description of SPR.

1. Identify initial set of revisions using RIO techniques.
2. Use evaluation over the instances to filter out poorly evaluated revisions.
3. Use rule structure similarity metric to filter revisions that would increase the structural complexity of the rulebase.
4. Revise rulebase using one of the remaining randomly selected revisions.
5. Return revised rulebase

some partition of the revision problem space. Together, the RIO techniques cover most of the revision problem space.

Figure 4.1 shows some of the revision problem space features described in Chapter 3. Figure 4.2 shows a partitioning of the revision problem space. Each partition (leaf in the DAG) of the revision problem space is labeled by the RIO techniques that are able to identify revisions in that partition. Some of the RIO techniques overlap in terms of the areas of the revision problem space that they cover. Some partitions are covered by only a single RIO technique.

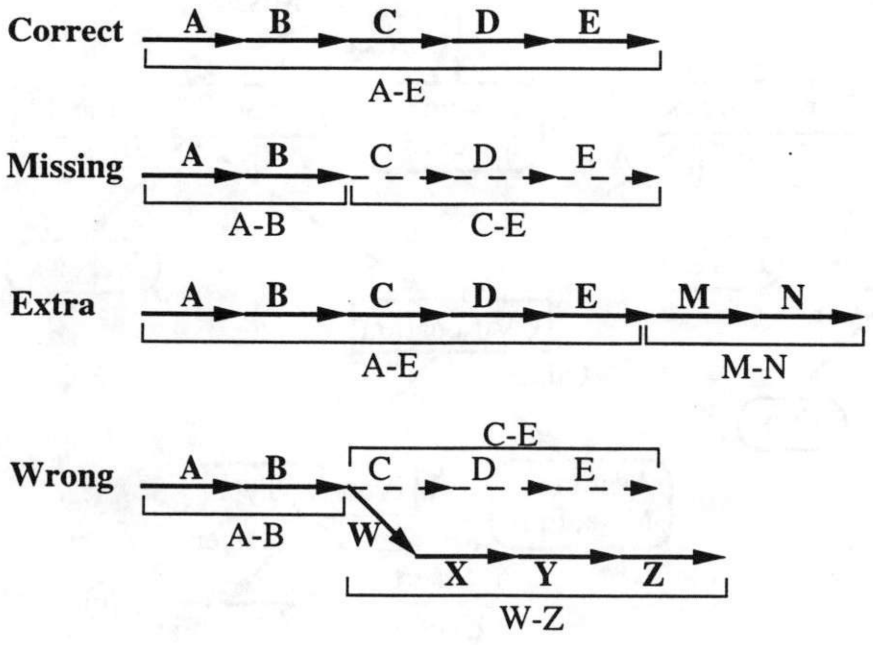
Of the RIO techniques associated with each partition in Figure 4.2, the highlighted technique (the first technique listed at each leaf) is the RIO technique that covers the entire partition. The other listed techniques may identify the correct revision, but they are not designed to cover the entire partition. For example, when the rule firing sequence deviation class is wrong, the problem location is `W` and the CV location is `W-Z`, the two listed techniques are ASSUME and EXP. ASSUME is the only RIO technique that is designed to identify revisions in the entire partition. EXP has the capability of happening upon the correct revision, but not for all problem types.

The actual product of each RIO technique is an ordering of high-level revisions. Revisions are ordered such that the revisions expected to produce the greatest decrease in error over the instances are placed highest in the ordering.

Each high-level revision produced by an RIO technique is associated with one or more of the low-level revisions that are used to directly revise the rulebase. The high-level revisions identifiable by each of the RIO techniques are shown in Table 4.3. None of the RIO techniques are able to identify all possible high-level revisions and some RIO techniques are able to identify the same high-level revisions. The expansion of high-level to low-level revisions is described in Section 4.3.3.

The product of all RIO techniques is a single ordering of high-level revisions. The single ordering is produced by interleaving the orderings produced by each of the individual RIO techniques. The interleaving process starts at the tops of the individual orderings and works down so that revisions at the top of the individual orderings are near the top of the merged ordering. Duplicates revisions lower in the merged ordering are removed.

The information used to identify and order revisions varies between RIO techniques. All techniques use some aspect of the trace information produced by evaluating each of the instances, and one technique uses simple experiments. The trace information is produced by running the instances over the unrevised rulebase. Recorded in each trace is the sequence of rule activation firings, the fact-lists before



Rule firing sequence deviation classes:
Correct, Missing, Extra, Wrong

Problem locations:
A-E, A-B, C, M, W

Constraint violation locations:
A-E, A-B, C-E, M-N, W-Z

Figure 4.1: Revision problem space features and possible values that were previously described in Chapter 3. These features are used to produce the revision problem space partitioning shown in Figure 4.2.

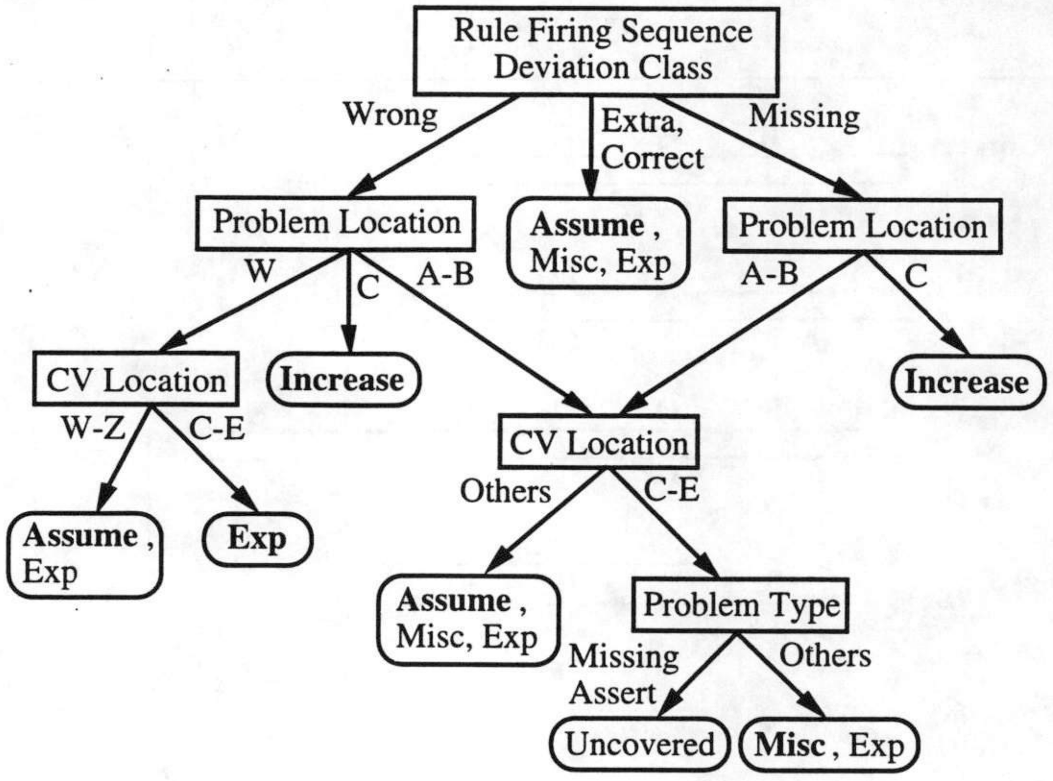


Figure 4.2: A partitioning of the revision problem space using features that were previously described in Chapter 3. Each partition is labeled with the RIO techniques that can identify appropriate high-level revisions for that partition.

Table 4.3: High-level revisions identifiable by each RIO technique. The mapping between high-level and low-level revisions is shown in Table 4.13.

RIO Techniques	**High-level Revisions**
Assume	Decrease Rule Add/Delete Action
Exp	Decrease Rule Delete Action
Increase	Increase Rule
Misc	Add Retract Delete Action

Table 4.4. High-level description of ASSUME.

```
ASSUME(traces)
  {
    for each trace in traces
      for each cv in constraint_violations(trace)
        revisions = revisions ∪ ASSUME_Find(cv, trace)
    revisions = ASSUME_Order(revisions)
    return revisions
  }
```

and after each firing, the contents of the agenda before and after each rule firing, and bindings that relate facts with satisfied CE, asserts with asserted facts and retracts with the facts that they retracted. In the following, the term *errorful trace* will refer to a trace associated with an instance with one or more CVs.

Two statistics, produced from trace information, that are used to order revisions, are *global error rate* and *execution count.* A global error rate is associated with each rule and is the ratio of CVs to constraints across all instances that execute the rule. For example, if a rule is executed by a set of instances and across those instances there are a total of E CVs from T constraints, the global error rate for the rule is $\frac{E}{T}$. Execution count is also associated with each rule and is the number of times that a rule is executed by all instances.

In the remainder of this section, each of the RIO techniques is discussed and demonstrated using examples.

Assume

The ASSUME RIO technique is similar to the technique used by A3, CLARUS, and FORTE at identifying points in the theory that should be revised. Unlike these other systems, however, ASSUME can only identify revisions to rules that fire during the execution of an instance. The task of identifying revisions to rules that should have fired but did not fire is left to the RIO technique INCREASE. The information used by ASSUME to identify revisions is errorful traces. The high level revisions identifiable by ASSUME are decrease-rule and add/delete action.

Table 4.4 shows a high level description of ASSUME. Each CV is used by ASSUME_FIND to identify a set of candidate revisions. ASSUME_ORDER uses the

Table 4.5. ASSUME single CV revision identification algorithm.

```
ASSUME_Find(cv, trace)
  {
    if cv is missing condition
      cond = missing_condition(cv)
      ASSUME_Missing_Cond(cond, trace)
    else /* cv is extra fact */
      fact = extra_fact(cv)
      ASSUME_Extra_Fact(fact, trace)
    return revisions
  }
```

Table 4.6. ASSUME missing condition revision identification algorithm.

```
ASSUME_Missing_Cond(cond, trace)
  {
    for each previously fired rule
      for each retract in rule that retracted a fact that matches cond
        add revision to delete retract from rule
        ASSUME_Extra_Rule(rule, trace)
      add revisions to add an assert for cond to rule
    return revisions
  }
```

Table 4.7. ASSUME extra fact revision identification algorithm.

```
ASSUME_Extra_Fact(fact, trace)
  {
    if fact was asserted by assert in rule
      add revision to delete assert from rule
      ASSUME_Extra_Rule(rule, trace)
    for each CE from a previously fired rule that was satisfied by fact
      add revision to add retract for CE to rule
    return revisions
  }
```

Table 4.8. ASSUME extra rule firing revision identification algorithm.

```
ASSUME_Extra_Rule(rule, trace)
  {
    add revision to decrease rule
    for each CE in rule
      if positive pattern CE
        fact = fact_that_satisfied(CE)
        ASSUME_Extra_Fact(fact, trace)
      else if negative pattern CE
        cond = CE_pattern(CE)
        ASSUME_Missing_Cond(cond, trace)
    return revisions
  }
```

resulting sets of revisions to produce the ordering of revisions that is returned by ASSUME.

ASSUME_FIND uses a single CV to identify a set of candidate revisions (see Table 4.5). It calls one of two algorithms depending on the type of CV. If the CV is caused by an unsatisfied condition (missing fact) in the final fact-list, ASSUME_MISSING_COND is called (see Table 4.6). If the CV is caused by an extra fact in the final fact-list, ASSUME_EXTRA_FACT is called (see Table 4.7).

ASSUME_MISSING_COND and ASSUME_EXTRA_FACT, along with a third algorithm ASSUME_EXTRA_RULE, recursively call one another while traversing the trace of the instance associated with the CV. Traversal of the trace starts at the final fact-list and works back toward the first rule execution.

$$\frac{CV\_Count}{Constraint\_Count} \times \log_2(Constraint\_Count) \tag{4.1}$$

ASSUME_ORDER uses the sets of candidate revisions produced for each CV to create a single ordered set of revisions. The set of revisions that ASSUME_ORDER orders is formed from the union of the individual sets of revisions. Revisions are sorted in descending order using the metric shown in equation 4.1. *CV_Count* is the number of CVs that suggest (add) the revision. *Constraint_Count* is the sum of all constraints associated with instances for which the rule associated with the revision fired. *Constraint_Count* is the maximum value possible for *CV_Count*, and may differ for different rules. The ratio, *CV_Count/Constraint_Count*, ranges between 0 and 1 and is highest when all of the constraints associated with a rule

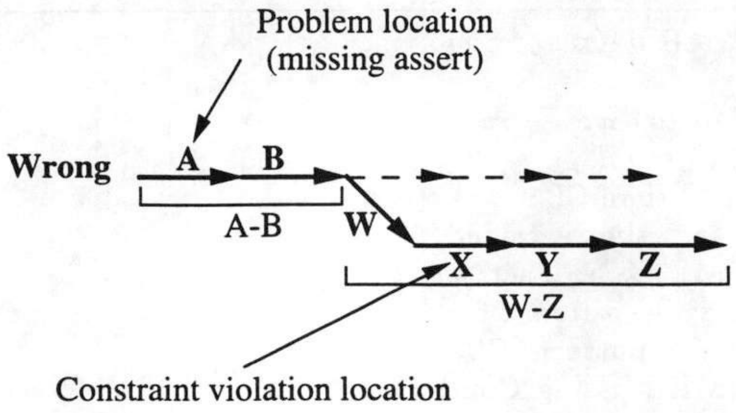


Figure 4.3. Example revision problem for ASSUME RIO technique.

firing are CVs. The product term $\log_2(Constraint\_Count)$ gives added evaluation to revisions that cover more constraints.

An Example

Assume that Figure 4.3 represents the rule activation firing sequence for some errorful instance. Further, assume that rule `X` asserts a fact that causes an extra fact CV and rule `A` is the problem rule because it has a missing assert action that allows rule `X` to fire. Based on these assumptions, the table below shows values for the relevant revision problem space features.

- Rule firing sequence deviation class is Wrong.
- CV location is W-Z (specifically rule Y).
- Problem location is A-B (specifically rule A).

Using the revision problem space tree shown in Figure 4.2, ASSUME, MISC and EXP are the RIO techniques that cover this partition. However, only ASSUME can handle this particular problem because EXP and MISC are not capable of generating add-assert revisions.

ASSUME proceeds to identify repairs for this instance and CV by calling ASSUME_FIND which in turn calls ASSUME_EXTRA_FACT with arguments, `F` (the extra fact) and `A-Z` (the rule firing trace for the instance). Recognizing that rule `X` asserted F, a revision to delete the assert action that asserted F in rule X is incorrectly identified. Correctly assuming that the assert in rule X is correct, but that rule X should not have fired when it did, a call is made to ASSUME_EXTRA_RULE with arguments, rule X and trace A-X.

When called, ASSUME_EXTRA_RULE identifies a revision to decrease-rule X because it incorrectly assumes that rule X may be under constrained and in need of, for example, specialization. Under the correct assumption, however, that rule X is not the problem and only fired because of a problem that occurred earlier, a call is made to either ASSUME_EXTRA_FACT or ASSUME_MISSING_COND for each of rule X's pattern CEs. ASSUME_EXTRA_FACT is called for each positive pattern CE because it assumes that the fact used to satisfy the CE should not been present in the fact-list. ASSUME_MISSING_COND is called for each negative pattern CE because it assumes that a matching fact should have been present in the fact-list to disable the firing of rule X.

A negative pattern CE in rule X causes ASSUME_MISSING_COND to be called with arguments C (the CE's condition) and trace A-W. Given that there are no retracts used by rules in the sequence A-W, the only revisions identified are revisions to rules A, B and W that add an assert for facts that would match C. Adding an assert to rule A is correctly identified as a revision.

When ASSUME_MISSING_COND decides to identify an add-assert revision for a rule, it may actually add multiple revisions (some that use variables in the rule). For example, if the missing condition is `(repair add-gas)`, it will first identify a revision to add `(assert (repair add-gas))`. If ?X and ?Y are variables introduced by positive pattern CEs in the rule, it will also add revisions to add `(assert (repair ?X))` and `(assert (repair ?Y))`.

For this instance and CV, ASSUME identifies many revisions, one of which is the correct revision. In general, ASSUME is used to identify revisions across all instance-CV pairs. When the pair corresponds to a partition handled by ASSUME, ASSUME will identify a set of revisions that includes the correct revision. When the pair does not correspond to a partition handled by ASSUME, ASSUME will not generate a set of revisions that necessarily includes the correct revision.

If all instance-CV pairs are appropriate for use by ASSUME, ASSUME_ORDER will place the correct revision at or near the top of the ordering. Recall that ASSUME_ORDER places revisions that are identified by more pairs higher in the ordering. When ASSUME uses some inappropriate pairs to identify sets of revisions, ASSUME_ORDER may place revisions that spuriously appear often in the sets of identified revision above the correct revisions. To the extent that more of the pairs are appropriate for ASSUME, ASSUME_ORDER will produce a better ordering.

Table 4.9. High-level description of EXP.

```
EXP(traces)
  {
    for each errorful trace
      revisions = revisions ∪ EXP_Find(trace)
    revisions = EXP_Order(revisions)
    return revisions
  }
```

Exp

The RIO technique EXP is designed to identify rules and actions that should not have fired. The technique involves running single-instance experiments. The high-level revisions returned by EXP are decrease-rule and delete-actions.

Table 4.9 shows a high-level description of EXP. For each errorful trace, a set of high-level revisions is identified using the algorithm EXP_FIND (see Table 4.10). In EXP_FIND, the instance associated with the trace is repeatedly executed over the rulebase. Once for each rule activation in the trace, the instance associated with the trace is executed up to but not including the activation, the activation is removed from the agenda and execution proceeds. Not allowing the rule activation to fire at the point that it would normally fire, approximates the effect of decreasing the rule and/or deleting an action from the rule. The benefit of the revision is measured as the difference between the number of CVs associated with the experiment and the number of CVs associated with the original trace of the instance. Only experiments that show improvement form revisions. If the same rule fires repeatedly in a trace, the experiment is run for each firing of that rule. The maximum evaluation across multiple firings of a rule, during a particular call to EXP_FIND, is the evaluation associated with the revision of that rule.

EXP_ORDER returns an ordered set of revisions. Revisions are sorted lexicographically. The primary sort key is the cumulative benefit (decrease in the number of constraint violations across all errorful instances) returned by EXP_FIND. The secondary sort key is the global error rate for the rule associated with the revision. The primary sort key places revisions that have been shown, via experiments, to decrease the number of CVs greatest near the top of the ordering. The secondary sort key orders revisions of more errorful rules higher.

Table 4.10. EXP single errorful instance revision identification algorithm.

```
EXP_Find(trace)
  {
    instance = instance(trace)
    for each fired rule
      execute instance up to fired rule
      remove fired rule from agenda
      execute instance to completion
      new_error = evaluate revised execution
      if new_error < error(trace)
        record revision to decrease fired rule
        record revisions to delete each action in fired rule
    return revisions
  }
```

An Example

Assume that Figure 4.4 represents the rule activation firing sequence for some errorful instance. Further, assume that rule W is overly general which allows it to fire instead of rule C (the rule that should have fired). Also, assume that the location of the CV is C-E (some rule in C-E did not execute an action that should have been executed to avoid the CV). Based on these assumptions, the table below shows values for the relevant revision problem space features.

- Rule firing sequence deviation class is Wrong.
- CV location is C-E.
- Problem location is W.

ASSUME and EXP are the only RIO techniques that have the capability of identifying decrease-rule revisions. However, based on the revision problem space tree shown in Figure 4.2, EXP is the only RIO technique that covers this partition.

ASSUME may accidently stumble upon the correct revision, but is not guaranteed to do so because the CV-location is an unfired rule. ASSUME will assume that the missing action execution caused by an unfired rule in the CV-location C-E is missing from a rule that fired. If the CV is an extra fact in the final fact-list that should have been retracted by a rule in C-E, it will incorrectly assume that a retract action is missing from some rule in A-Z, and that the action and rule that

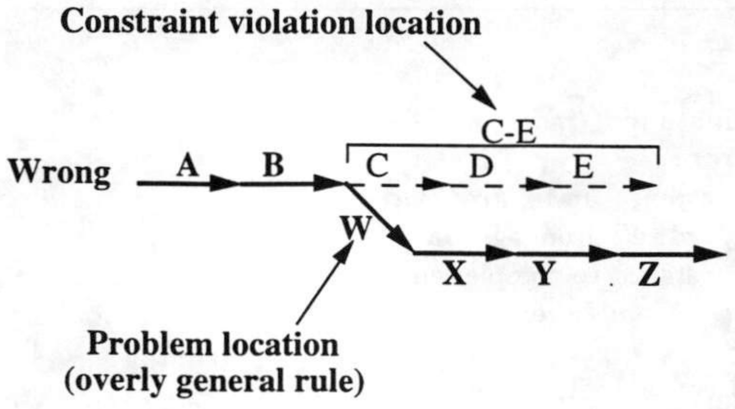


Figure 4.4. Example revision problem for EXP RIO technique.

asserted the extra fact should never have executed. If the CV is a missing fact in the final fact-list that should have been asserted by a rule in `C-E`, it will incorrectly assume that an assert action should be added to a fired rule.

Given that the instance associated with the revision problem shown in Figure 4.4 had only the single CV, EXP would call EXP_FIND to identify revisions associated with this instance. The first experiment it would run would be to remove the activation associated with the execution of rule A, complete the run and evaluate the number of associated CVs. If the number of CVs produced by the experiment is less than the number of CVs produced by a normal run of the instance, a revision for rule `A` would be identified. An experiment where activation `B` is removed would be executed next.

When the experiment associated with the removal of activation `W` is performed, the correct execution of rules `C` through `E` should occur. This will lead to a decrease in the number of constraint violations and the identification of a decrease-rule revision for rule `W`. Note, whenever EXP identifies a decrease-rule revision for a rule, it also identifies delete-action revisions for that same rule.

EXP is a heuristic RIO technique because it only approximates the effect of decreasing a rule or deleting one of its actions. In the above example, the benefit of removing an activation for rule `W` may be nullified by the appearance of another incorrect execution of rule `W` later in the execution.

Actually, the only partition of the revision problem space that EXP is designed to cover is the partition of the revision problem space associated with this

Table 4.11. High-level description of INCREASE.

```
INCREASE(traces)
  {
    for each rule in rulebase
      add revision to increase rule
    revisions = INCREASE_Order(revisions)
    return revisions
  }
```

example. The decrease-rule revision is the only type of revision that EXP needs to be able to identify in order to cover this partition. The identification of delete-action revisions with each decrease-rule revision was enabled because this technique has the natural capability of identifying actions that should never have fired.

Note, unlike ASSUME, EXP does not use the individual CVs to identify revisions, it uses a count of the number of CVs that an instance has before and after experiments.

Increase

The INCREASE RIO technique identifies rules that need to be increased (see Table 4.11). The low-level revisions associated with the increase-rule high-level revision include increasing the salience of the rule and deleting single CEs from the rule. Evaluating an INCREASE high-level revision tends to be cheap because the average number of CEs per rule is usually small and there are few different salience values used in a rulebase.

The difficulty in trying to intelligently identify increase-rule revisions and the low cost required to evaluate these revisions led to the decision of making INCREASE a brute force approach. It identifies an increase-rule revision for each rule in the rulebase.

The most common approach used by Horn clause based revision systems for performing the analogous task of identifying rules that need to be generalized is to identify, while backward chaining, those predicates that can provide the needed assertion. A similar approach is possible with production rules by looking at the assert actions in a rule. Unfortunately, the partition of the revision problem space covered by such an approach would be small and other RIO techniques would still be required to identify increase-rule revisions.

Table 4.12. High-level description of MISC.

```
MISC(traces)
  {
    for each rule in rulebase
      add revision to delete each action from rule
      add revision to add a retract for each pattern CE in rule
    revisions = MISC_Order(revisions)
    return revisions
  }
```

As an alternative, more comprehensive approach, a technique like ASSUME (or simply an extension to ASSUME) could be used to identify rules that could provide needed assertions. Such an approach would both generate revisions under the assumption that the identified rule should be increased and identify revisions under the assumption that the identified rule is correct but that some problem earlier in the run kept the rule from firing (possibly some other rule that should be increased). Unfortunately, experiments showed that this approach, in the presence of multiple asserts per rule and retract actions, tends to identify most or all of the revisions already identified by INCREASE.

In terms of computational expense, the number of revisions identified by INCREASE is linear to the number of rules in the rulebase and the average number of CEs per rule.

INCREASE_ORDER returns an ordered set of revisions. Revisions are sorted lexicographically. The primary sort key is the rule execution count. The secondary sort key is the global error rate for the rule. The primary sort key places revisions that fire rarely near the top of the ordering. The secondary sort key orders revisions of more errorful rules higher.

Misc

The MISC RIO technique, like the INCREASE RIO technique, is a brute force method for identifying revisions (see Table 4.12). It identifies all possible delete-action (assert and retract) and add-retract revisions in the rulebase. The evaluation of a MISC revision tends to be cheap because there are usually few asserts, retracts and pattern CEs in a rulebase. Like INCREASE, the number of revisions generated by MISC is linear to the number of rules in the rulebase and the average sum of the number of assert and retract actions and positive pattern CEs in a rule.

The MISC RIO technique is used to cover areas of the revision problem space not covered by any of the other RIO techniques. The MISC RIO technique is required when the CV location corresponds to rules that did not fire, `C-E`, and the problem location is in the region `A-B`. Revisions in this part of the space are hard to identify because there is no traceable sequence of actions (and inactions) that lead from the occurrence of the problem to the CV.

MISC_ORDER returns an ordered set of revisions. Revisions are ordered by the rule's global error rate. The most errorful rules are placed highest in the ordering. After INCREASE_ORDER, MISC_ORDER is the weakest of the ordering techniques.

Uncovered Territory

The only partition of the revision problem space that is not covered by an RIO technique is the partition labeled `Uncovered` in Figure 4.2. This region corresponds to a partition similar to the partition that MISC is designed to cover, except that the problem type in this region is missing assert. An approach similar to MISC could have been used to heuristically cover this partition. Unfortunately, an approach that blindly adds asserts to rules without knowing anything about the patterns to add would be prohibitively expensive. There are too many possible patterns that could be used to form asserts.

4.3.2 RIO Overview

All RIO techniques work together to identify an initial ordering of candidate revisions. Given the assumption that the rulebase has only a single problem, for some instance-CV pairs, some of the RIO techniques will be inappropriately used to identify revisions. Over all instance-CV pairs, a distribution of revision problems could show which RIO techniques are most important toward identifying the correct revision given the available instances. An analysis of the benefit of each RIO technique in the context of different revision problem distributions is presented in Chapter 5.

Table 4.13. High-level/low-level revision mapping.

High-Level Revisions	Low-level Revision Classes
Decrease Rule	Add CE (Specialize Rule) Decrease Salience Delete Rule
Increase Rule	Delete CE (Generalize Rule) Increase Salience
Add Assert	Add Assert
Add Retract	Add Retract
Delete Assert	Delete Assert
Delete Retract	Delete Retract

4.3.3 High-Level Revision Expansion & Evaluation

The second step after the initial identification of a set of high-level revisions by the RIO techniques is to expand the high-level revisions into low-level revisions that are each evaluated over the instances. During evaluation, only the best evaluated low-level revisions are retained. This step constitutes a filtering of the candidate revisions through the use of evaluation.

The actual expansion and evaluation of revisions are done in an interleaved manner. One-by-one, and in order, high-level revisions are passed to a set of revision-specific operators. Each operator takes the high-level revision and serially generates low-level revisions that are each evaluated over the instances. Table 4.13 shows the mapping between high-level and low-level revisions.

The evaluation of a low-level revision is accomplished by temporarily making the revision to the rulebase and then evaluating over the instances. Some low-level revisions do not require evaluation over all instances, so only instances affected by the modification to the rulebase are re-evaluated. During the evaluation of low-level revisions, a set of best evaluated revisions is maintained. All revisions in the set have the same evaluation.

Evaluation of the revisions over the instances is the most time consuming step in the operation of SPR. The default behavior of SPR is to evaluate each of the high-level revisions identified by the RIO techniques.

When time is constrained, SPR can be instructed to halt the expansion and evaluation of revisions after a limit on the number of low-level revision evaluations is reached. The current set of best evaluated revisions is passed on. Since the ordering of high-level revisions heuristically places the correct revision near the top of the ordering, evaluation of all identified high-level revisions tends to be a case of diminishing returns. An empirical evaluation of the use of this form of resource bounding is presented in Chapter 5.

Decrease Rule

The decrease-rule high-level revision is expanded using operators that specialize the rule, decrease the rule's salience and delete the rule. The specialize-rule operator creates low-level revisions that add a single pattern or test CE to a rule. The decrease-salience operator creates low-level revisions that decrease the salience of the rule. The patterns used to form pattern CEs and the cut-off values used to form test CEs are generated using heuristic techniques that take advantage of trace information and the product of simple experiments.

Pattern CEs are formed from generalizations of facts found in certain fact-lists. The fact-lists that these facts are extracted from are the fact-lists that allowed activations of the rule being revised to fire. Two different sets of facts are extracted from these fact-lists. One set comes from fact-lists that are associated with appropriate firings of the rule being revised. The other set of facts comes from fact-lists that are associated with inappropriate executions of the rule.

Those facts, extracted from fact-lists associated with inappropriate firings of the rule, are referred to as *bfacts* and are used during the construction of negative pattern CEs. The bfacts represent patterns that should disable the firing of the rule by not matching negative pattern CEs produced using the facts. The other set of facts is referred to as *gfacts* and is used to created positive pattern CEs. These facts are associated with fact-lists for which it is acceptable to fire the rule.

The identification of fact-lists used to form bfacts is done during the identification of decrease-rule revisions. During decrease-rule identification, fact-lists associated with rule activations found in the trace information are marked when the rule associated with the activation is used to identify a decrease-rule revision. The identification of an initial set of fact-lists used to form gfacts is done in a similar manner. Fact-lists associated with activations found in trace information for the rule being revised that are not marked as having been used to identify a decrease-rule revisions are used to form gfacts.

Additional fact-lists, used to produce more gfacts, are created by running inexpensive experiments that allow previously inappropriately fired rules to fire appropriately. Experiments similar to those run by Exp are run using traces that contain marked rule activations for the rule being revised. For each marked activation of the revised rule, the instance associated with the trace is run up to the firing of the activation, the activation is moved to another place on the agenda, and execution continues. If a search of the remainder of the rule executions, identifies a firing of the rule, the fact-list associated with the new rule activation is used to produce more gfacts.

The patterns used to produce CEs from a set of facts are formed from the LGG (Popplestone, 1970) of subsets of the facts. Added CEs may include both variables and constants.

Three types of test CEs are used to specialize rules. Two of the types perform variable-constant comparisons, e.g. ?X > 5. The third type performs variable-variable comparisons, e.g. ?X = ?Y. The variables used to form test CEs are extracted from the positive pattern CEs found in the rule. For variable-constants comparisons, the possible cut-off values are extracted from the gfacts and bfacts produced to generate pattern CEs. Midpoints between adjacent values in an ordering of the set of values are used as cut-off values.

The decrease-salience operator identifies a minimal set of alternative salience values that have the potential of decreasing the rule when evaluated over the instances. It starts by identifying all saliences in the rulebase that are less than the current salience of the rule. To that set it adds the current salience of the rule and the minimum possible salience (-10000). Lastly, it orders these saliences and adds saliences that are half-way between each adjacent pair of saliences in the ordering. These saliences, less the original salience of the rule and the minimum salience, are used to produce low-level decrease-salience revisions. For example, if the rule's original salience is -10 and the other lesser saliences in the rulebase are -20 and -100, the saliences used to form the low-level revisions would be -5050, -100, -60, -20, and -15.

The re-evaluation of decrease-rule low-level revisions is only done over a subset of the instances. The evaluation of instances that never executed the rule that is being revised will not be affected by this type of revision.

Increase Rule

The increase-rule high-level revision is expanded using operators that generalize the rule and increase the rule's salience. Each generalize-rule low-level revision corresponds to the deletion of a single CE.

The increase-salience operator identifies a minimal set of alternative salience values that have the potential of increasing the rule when evaluated over the instances. It starts by identifying all saliences in the rulebase that are greater than the current salience of the rule. To that set it adds the current salience of the rule and the maximum possible salience (10000). Lastly, it orders these saliences and adds saliences that are half-way between each adjacent pair of saliences in the ordering. These saliences, less the original salience of the rule and the maximum salience, are used to produce low-level increase-salience revisions. For example, if the rule's original salience is -10 and the other greater saliences in the rulebase are 0 and 10, the saliences used to form the low-level revisions would be -5, 0, 5, 10 and 5005.

All of the low-level revisions produced from an increase-rule revision are re-evaluated over all instances because increasing a rule may cause the rule to fire under circumstances where it originally did not fire.

Add/Delete Action

Each high-level revision to add or delete an action (assert or retract) expands to only a single low-level revision. If the revision suggests the addition of an action, the action is added to the LHS of the rule and the revised rulebase is evaluated over the instances. If the revision suggests the deletion of an action, the action is deleted and the rulebase is evaluated. Add/delete action low-level revisions need only be re-evaluated over instances that originally executed the rule.

4.3.4 Rulebase Structure Filtering

The last step prior to the final selection of a revision is the filtering out of revisions that would produce ill-structured rules. Rule structure filtering biases the reviser toward selecting revisions that keep/make rules similar to other rules in the rulebase. The empirical analyses in Chapter 7 demonstrate that this bias is good at improving rulebase accuracy.

Rule structure filtering looks at the rule being revised and identifies the best group of rules consistent with the rule. It then revises the rule and again identifies the best group consistent with the rule. If the quality of the revised rule's best group is worse than the quality of the unrevised rule's best group, the revision is removed. If after filtering, no revisions remain, the original set of revisions is returned, otherwise only the unfiltered revisions are returned.

Rules and groups of rules are described in terms of a set of feature-value pairs. All rules have the same number of feature-value pairs. A rule belongs to a group of rules if its feature-value pairs are a superset of the feature-value pairs that describe the group.

The quality of a group of rule is a function of the number of rules in the group and the number of features used to describe the group. The larger the number of rules and size of the description, the better the quality of the group. The expression used to calculate group quality is $Description\_Size + \log_2(Group\_Size)$, where $Description\_Size$ is the number of feature-value pairs that describe the group and $Group\_Size$ is the number of rules in the group. Two groups have the same quality if a linear decrease in the number of feature-values is offset by an exponential increase in the number of rules covered by the group.

4.4 Other Revision Constraints

In addition to final fact-list constraints and rule execution limit constraints, SPR is able to accept other forms of revision constraints. These constraints are hard constraints and are used to filter the number of legal revisions. They include variable typing, constraints on the types of facts that may be assertable by a rule, and constraints on the types of revisions that may be used to revise the rulebase.

Variable typing constraints allow variables to be typed as either a number, a symbol or anything (default). They are used to reduce the number of add-CE and add-assert revision evaluations. Typing constraints are represented by patterns that constrain the variables in pattern CEs that are subsumed by the patterns. For example, the variable constraint `(age :symbol :number)` would constrain the variable `?X` in `(age sean ?X)` to be a number.

Another type of constraint accepted by SPR is a constraints on the type of facts that can be assertable by a rule. For example, the non-assertable fact constraint `(sex ?X ?Y)` declares that `(assert (sex shannon ?Z))` is an illegal revision.

SPR includes two additional constraints that disallow certain types of revisions. For example, the no-retracts constraint disallows add-retract revisions. The do-not-change-salience constraint disallows revisions that would modify the salience of rules. The purpose of these two constraints is to narrow the revision space for rulebase revision problems where it is obvious that these types of revisions are not needed.

4.5 Chapter Summary

This chapter described CR2, an implemented system that is capable of revising production system rulebases. CR2 is organized as a hill-climbing system with main component SPR. SPR is designed in the context of the revision problem space model described in Chapter 3. It is designed to revise rulebases that have only a singleton problem. Through repeated executions of SPR, CR2 is able to revise rulebases that may contain multiple problems. SPR and its components the RIO techniques and rule structure filtering are described.

The RIO techniques identify and order sets of high-level revisions that are expanded and evaluated over the instances. Each RIO technique is designed to identify revisions associated with specific partitions of the revision problem space. Most, but not all, of the revision problem space is covered by RIO techniques. An empirical analysis of the utility of each RIO technique is described in Chapter 5.

Rule structure filtering takes as input the set of best evaluated revisions and removes revisions that would produce ill-structured rules. Rule structure filtering is a bias toward revisions that keep/make rules similar in structure to other rules in the rulebase. An empirical analysis of the utility of rule structure filtering is described in Chapter 7.

Chapter 5
Singleton Problem Reviser: Results & Evaluation

5.1 Chapter Overview

SPR is designed in the context of a set of explicit assumptions. The most important of these assumptions concerned the kinds of problems that SPR would be expected to handle. Specifically, SPR is designed to revise rulebases that have only a single problem.

In this chapter, SPR is evaluated in the context of this assumption. All results are based on the use of SPR, not CR2, as the revision system. In other words, SPR is executed only once per experiment. Each experiment revises a mutated rulebase that is only one revision away from being correct. All mutated rulebases have the potential of being correctly revised using a single execution of SPR's revision operators.

The main goal of this chapter is to empirically validate the design of SPR. Using the revision problem space model described in Chapter 3 and the mapping between revision problems and RIO techniques described in Figure 4.2, a RIO distribution, associated with each mutated rulebase, is used to explain why certain mutated rulebase are more difficult for SPR to revise than others. Other goals include an empirical investigation of which problem types are most difficult for SPR to repair, and an understanding of the effect of limiting the number of low-level revision evaluations.

5.2 Experimental Methodology

Experiments were performed using three domains. Each of these domains has a rulebase and a set of instances. The instances are consistent with the rulebases (produce no CVs). In order to evaluate SPR, a set of mutated rulebases was formed from each correct rulebase. Mutated rulebases were formed by changing salience or by randomly adding or removing actions (asserts and retracts) and CEs. All mutated rulebases have only a single mutation and are guaranteed to be inconsistent with some constraints.

For each domain, sets of experiments were performed using the mutated rulebases. An experiment consisted of forming a train/test partition of the instances and then revising a mutated rulebase using only the training instances (train partition). CV error and elimit error, on both the training instances and test instances, were measured before and after revision. Experiments varied by the mutated rulebase revised, the training set size, rlimit (limit on the number of low-level revision evaluated), and the specific train/test partition of the instances.

The averages corresponding to the data points used to form graphs were computed using the results of at least 20 individual experiments. Confidence intervals are at two standard deviations (95.44%).

5.3 Domains

The domains used to evaluate SPR are the student loan, auto diagnosis and nematode identification domains. Each of these domains is different in terms of characteristics of their respective rulebases.

The student loan domain is the simplest domain. It has a rulebase of 15 rules and has 1000 instances. Each instance has 9 constraints. All rules use the same salience. Every rule has exactly one assert action and no other actions. No rules perform user queries. The correct execution of the rulebase generates the deductive closure of the facts in the initial fact-list. For a more detailed description of this domain see Appendix A. 50 mutated rulebases were formed for the student loan domain.

The auto diagnosis domain is more complex than the student loan domain. Unlike the student loan domain, the auto diagnosis rulebase uses multiple saliences. Some rules have multiple assert, bind, and printout actions. No rules have retract actions. Execution of the auto diagnosis rulebase does not generate any kind of deductive closure of the initial fact-list. This domain includes 27 rules and 250 instances. Each instances has at least 11 constraints. A more detailed description of the auto diagnosis domain is presented in Appendix B. 50 mutated rulebases were formed for this domain.

Table 5.1: Average initial error for each of the domain's mutated rulebase sets evaluated over all instances.

Domains	**CV Error mean (sd)**	**Elimit Error mean (sd)**
Student Loan	5.05e-2 (6.01e-2)	0.00e-0 (0.00e-0)
Auto Diagnosis	9.07e-2 (1.40e-1)	0.00e-0 (0.00e-0)
Nematode Identification	8.68e-3 (1.27e-2)	7.36e-3 (2.31e-2)

The nematode identification domain is by far the most complex of the three domains. Its rulebase uses multiple saliences, and its rules have assert, retract, bind and printout actions. Some rules have multiple actions. The nematode identification rulebase has 93 rules and 55 instances. There are at least 23 constraints for each instance. A detailed description of this domain is presented in Appendix C. 42 mutated rulebases were formed from the correct nematode identification rulebase.

The mean and standard deviation of the CV and elimit errors across the mutated rulebases for each domain are presented in Table 5.1. Only the mutated nematode rulebases produce elimit errors. Note, elimit error is the number of times, for all instances, that the activation execution limit was exceeded. Elimit was described in Section 2.3.2.

5.4 RIO Distribution

Given a rulebase with a singleton problem and a set of instances that produce CVs when evaluated using the rulebase, each CV-instance pair is associated with one or more elements of the revision problem space described in Chapter 3. Across multiple CV-instance pairs, each revision problem has an associated frequency of occurrence. Since the mutated rulebases described above satisfy the constraints of the revision problem space model, for each mutated rulebase, a revision problem frequency distribution was produced.

For this research, the importance of these distributions is in the generation of another type of distribution: the frequency of revision problems associated with each RIO technique. That is, each revision problem is associated with a leaf in the tree shown in Figure 4.2, and the coverage of each leaf is dependent on a RIO technique. Therefore, for each mutated rulebase, a RIO distribution was produced that describes the percentage of revision problems dependent on each

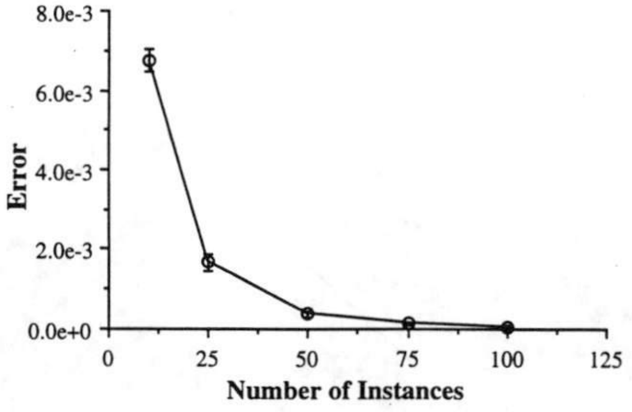


Figure 5.1: Average CV error of revised rulebases as a function of the number of training instances for the student loan domain.

RIO technique. These distributions are used to understand the results presented below.

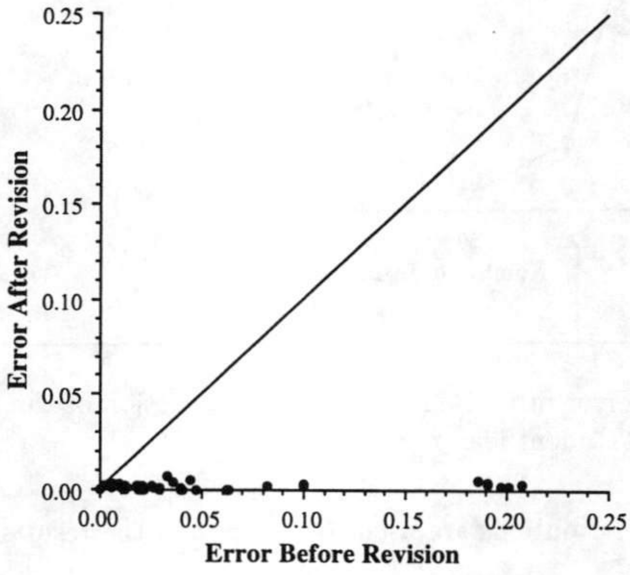
For example, assume that the problem type for a mutated rulebase is *extra action*, and 2 CV-instance pairs produce 3 revision problems, each with rule firing deviation class *extra*, and 7 CV-instance pairs produce 7 revision problems, all with problem location A-B and CV-location C-E. Then, by the tree in Figure 4.2, the first three revision problems correspond to a leaf that depends on ASSUME, and the other 7 revision problems correspond to a leaf that depends on MISC. The RIO distribution for this mutated rulebase is 30% ASSUME and 70% MISC.

5.5 Full Evaluation

The results presented in this section are based on experiments where all high-level revisions, identified by the RIO techniques, are expanded and evaluated. No resource limit is placed on the number of low-level revision evaluations. The main purpose of this section is to show which problem types are easy and difficult to repair and to show weaknesses in the coverage of the RIO techniques.

The first results are based on the student loan domain. Figure 5.1 shows, as expected, that CV error decreases as the number of training instances increases. This figure also shows that there are few CVs at 100 training instances.

A. 25 Training Instances



B. 100 Training Instances

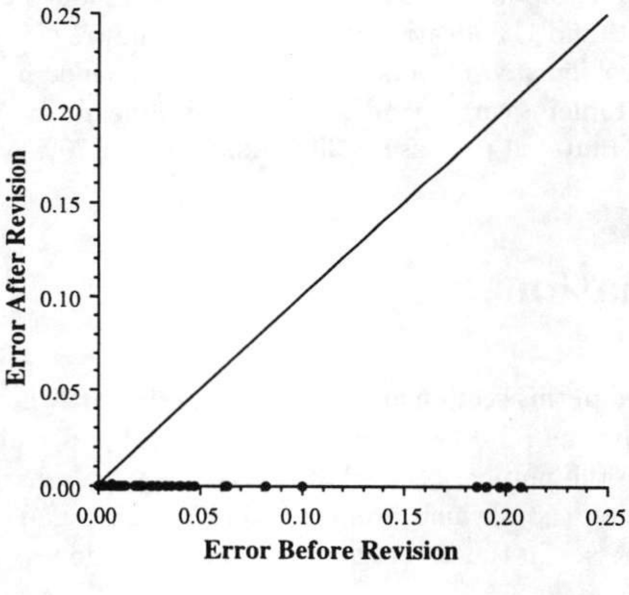


Figure 5.2: Average CV error after revision as a function of CV error before revision for each mutated student loan rulebase.

Table 5.2. Problem type error rates for student loan domain.

Problem Type	**25 Instances**	**100 Instances**
Missing CE	23.0%	1.0%
Missing Assert	0.0%	0.0%
Extra CE	0.6%	0.0%
Extra Assert	0.0%	0.0%

Figure 5.2 shows average error after-revision as a function of average error before-revision for each of the 50 mutated student loan rulebases. The first thing to note from these graphs is that all of the points are below the main diagonal. This indicates that for each mutated rulebase, the average after-revision error is less than the average before-revision error. At training set size 100, only 9 of the 50 mutated rulebases are not 100% accurate.

Further examination of the results of the individual experiments, used to form Figure 5.2a, show that SPR is actually doing much better than just described. Some of the results that were used to produce the averages were based on experiments where the training instances were consistent with the mutated rulebase. Only the test instances produced CVs. Unrepresentative train/test partitions of this kind occurred often with mutated rulebases for which there were few CVs and for experiments where training set size is small. When presented with such an unrepresentative set of training instances, SPR returns the unrevised rulebase.

When the results from unrepresentative training set experiments were omitted from the calculation of average errors, the number of non-zero after-revision errors at 25 training instances dropped from 40 to 16. More than half of the revision problems were solved at 25 training instances. The same recalculation at training set size 100, dropped the number of non-zero after-revision errors from 9 to 4.

Additional analyses of the individual experiments showed which problem types are hardest for SPR to repair. Table 5.2 shows that at 25 training instances, 23% of the individual experiments on mutated rulebases, that had a missing CE, produced non-zero after-revision errors. At 100 training instances, 1% of the missing CE experiments produced non-zero after-revision errors. The four mutated rulebases that produced non-zero after-revision errors at 100 training instances are all associated with mutated rulebases that have a missing CE and are in need of specialization. All other problem types associated with the mutated rulebases were easily solved by SPR.

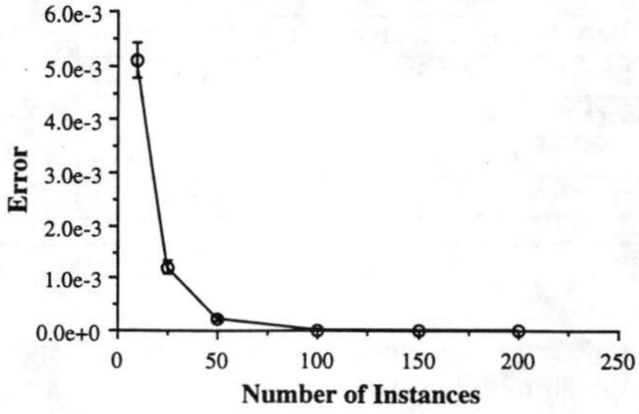


Figure 5.3: Average CV error of revised rulebases as a function of the number of training instances for the auto diagnosis domain.

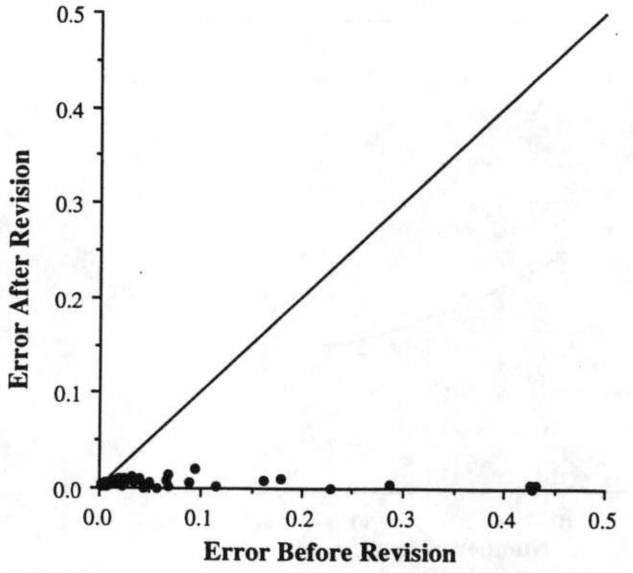
The next set of results to be discussed is based on the auto diagnosis domain. This domain is more difficult for SPR to revise because of the presence of multiple saliences. Figure 5.3 shows average CV error of the mutated auto diagnosis rulebases as a function of training set size. At 100 training instances, most CVs are solved.

Figure 5.4 shows a pair of scatter plots for the mutated auto diagnosis rulebases. CV error after revision is presented as a function of CV error before revision for each mutated rulebase. At 100 training instances, all after-revision errors are less than their respective before-revision errors. At 10 training instances, all but two after-revision errors are less than the before revision errors.

As with the student loan experiments, some of the auto diagnosis experiments are based on unrepresentative training sets. After removal of the results from these experiments, the number of non-zero average errors at training set size 10 goes from 47 to 35. At training set size 100, the number of non-zero average after-revision errors goes from 6 to 5.

An analysis of which problem types are hardest to repair for the auto diagnosis domain shows that missing CE is again the most difficult. At 10 training instances, 49.1% of the missing CE experiments produced non-zero after-revision errors (see Table 5.3). 4 of the 5 non-zero errors at 100 training instances are associated with missing CE mutations. Rule specialization is difficult because there are many ways that a rule may be specialized. At low training set sizes, it is even more difficult because more "add CE" revisions tend to have the same evaluation.

A. 10 Training Instances



B. 100 Training Instances

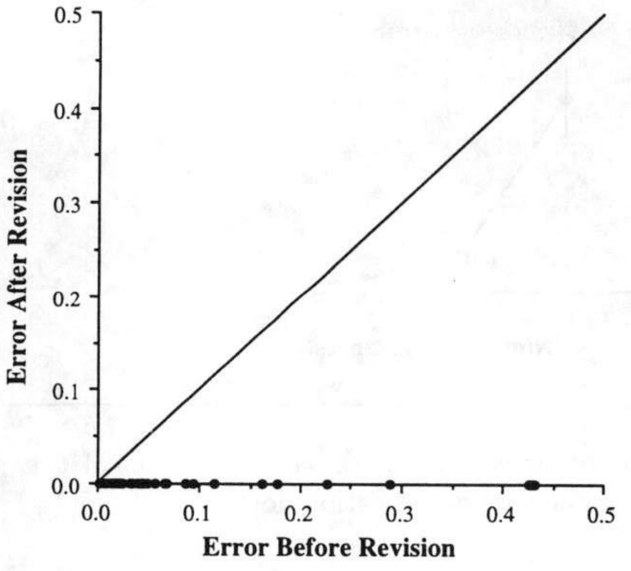
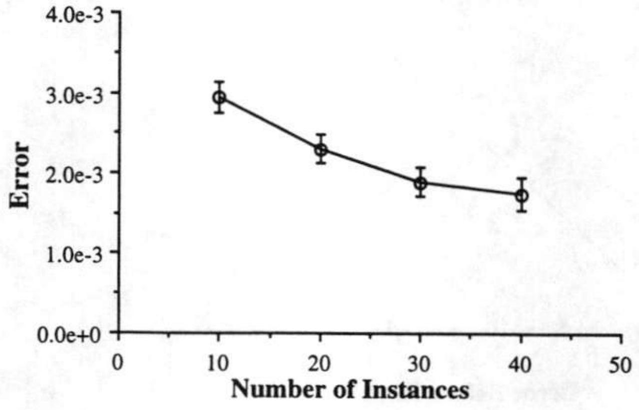


Figure 5.4: Average CV error after revision as a function of CV error before revision for each mutated auto diagnosis rulebase.

A. CV Error



B. Elimit Error

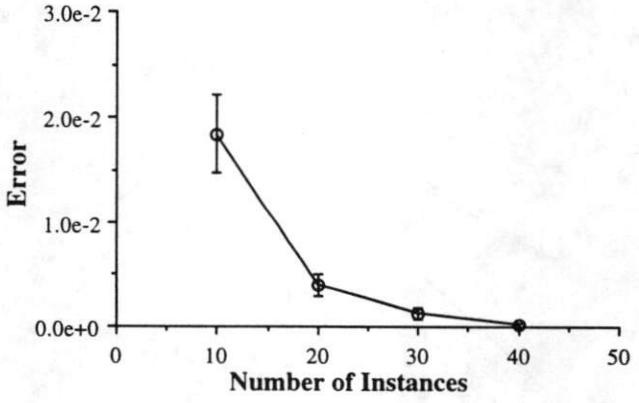


Figure 5.5: Average error of revised rulebases as a function of the number of training instances for the nematode identification domain.

Table 5.3. Problem type error rates for auto diagnosis domain.

Problem Type	10 Instances	100 Instances
Missing CE	49.1%	2.4%
Missing Assert	17.2%	0.0%
Extra CE	15.6%	0.0%
Extra Assert	3.2%	0.0%
Low Salience	20.7%	0.0%
High Salience	17.4%	0.0%

The last set of results to be discussed is based on the nematode identification domain. This domain is by far the most difficult domain for SPR to revise. Unlike both the student loan and the auto diagnosis domains, the nematode identification rulebase uses retract actions. In fact, the way retracts are used, makes many of the mutated nematode identification rulebases particularly difficult to revise.

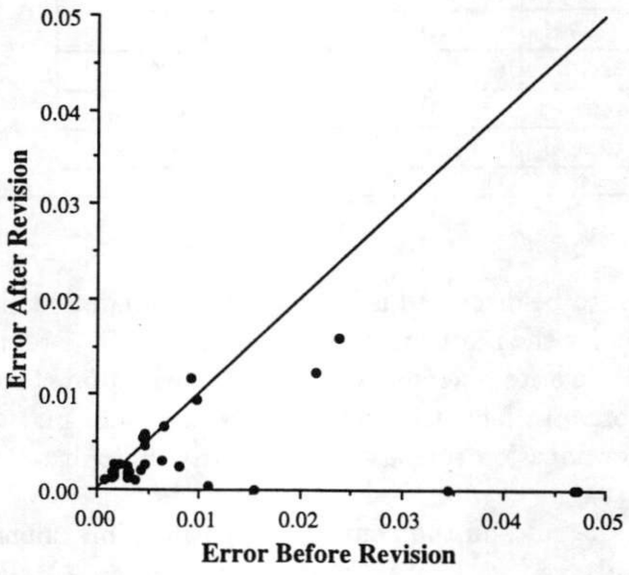
Specifically, most of the rules in the nematode identification rulebase have a retract for every positive pattern CE. That is, when a rule is executed, all facts that are used to activate the rule are retracted. This characteristic makes it difficult for a blame assignment techniques based on final fact-list constraints to identify misfired rules and actions.

This "retract-everything" characteristic caused another difficulty when trying to generate mutated rulebases. No CEs, without associated retracts, could be found, that when omitted, produced an inconsistent rulebase. Only mutated rulebases with missing CE/retract pairs produced CVs. Unfortunately, since mutated rulebases with missing CE/retracts cannot be corrected during a single execution of SPR, no mutated rulebases with only a missing CE are included in the 42 mutated nematode identification rulebases.

As evidence for the claim that the nematode identification domain is difficult to revise, Figure 5.5a shows CV error as a function of training set size. CV error does not come close to asymptoting to zero as it does for the other domains. Figure 5.5b demonstrates that, unlike the student loan and auto diagnosis domains, some mutated nematode identification rulebases produce elimit errors. At 40 training instances, elimit error is almost zero.

Figure 5.6 shows CV error after revision as a function of CV error before revision for each of the 42 mutated rulebases. Again, unlike the student loan and auto diagnosis domains, many of the mutated rulebases are not well revised, even

A. 10 Training Instances



B. 40 Training Instances

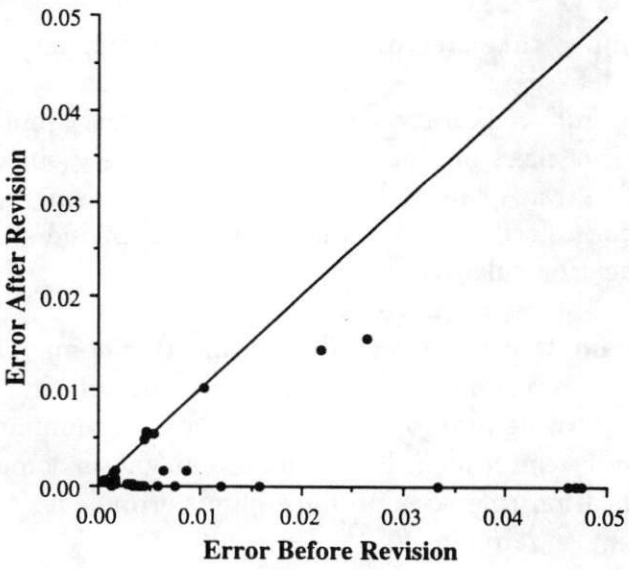


Figure 5.6: Average error after revision as a function of error before revision for each mutated nematode identification rulebase.

Table 5.4. Problem type error rates for nematode identification domain.

Problem Type	10 Instances	40 Instances
Missing Assert	69.2%	38.3%
Missing Retract	18.2%	0.0%
Extra CE	0.0%	0.0%
Extra Assert	5.1%	0.0%
Extra Retract	3.4%	1.1%

at 40 training instances, eight of these mutated rulebases produce average after-revision errors that are greater or equal to their average before-revision errors. Of course, most of the points at both training set sizes are well below the main diagonal. Removal of the results from unrepresentative experiments shows that only 22 (not 37) and 15 (not 27) of the mutated rulebases at training set sizes 10 and 40, respectively, produce non-zero after-revision errors.

For the nematode identification domain, an analysis of which problem types are hardest to repair shows a different result than for the other two domains (see Table 5.4). Missing assert is the most difficult to repair, 69.2% at 10 training instances and 38.3% at 40 training instances. However, looking back at Tables 5.2 and 5.3 for the student loan and auto diagnosis domains, respectively, the missing assert problem type is among the easiest to repair.

The explanation for this requires an examination of the RIO distributions for these rulebases. Fourteen of the fifteen mutated rulebases (training set size 40) that produced non-zero after-revision errors had missing assert problem types. Twelve of these fifteen rulebases had RIO distributions that show a 0% dependence on any RIO technique. They are all consistent with the uncovered leaf shown in Figure 4.2. Seven of the eight mutated rulebases that produced greater or equal after-revision error are included in the twelve 100% uncovered revision problems.

Since these uncovered revision problems are shown here to occur fairly often, not being able to identify revisions in this partition of the revision problem space is a significant problem. As the main reason for the uncovered partition is the sole reliance on final fact-list constraints, it seems that additional forms of constraints will be needed in order to, in general, solve the production system revision problem.

5.6 Partial Evaluation

The results presented in this section are based on experiments where a resource limit (rlimit) is placed on the maximum number of low-level revision evaluations. The purpose of this section is to show, empirically, some of the issues involved in using this kind of resource limit.

The first results were formed from experiments based on the student loan domain. Figure 5.7a shows CV error as a function of training set size for each of three rlimit settings. Except when the rlimit is set to 100, as training set size increases error tends to decrease. The greater the rlimit, the lower the asymptotic error.

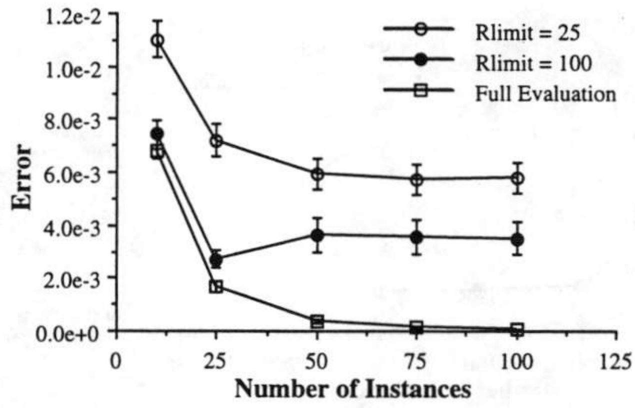
Figure 5.7a shows that when rlimit is 100, error decreases from 10 to 25 training instances and then increases slowly. This can be explained by noting that larger training set sizes tend to identify larger numbers of high-level revisions. For the student loan domain, many of the problem types are missing CEs and specialization of an average student loan rulebase rule requires the evaluation of roughly 100 low-level revisions. At 25 training instances, fewer than 100 revisions are required. At 100 training instances, more than 100 revisions are required. Therefore, when the training set size is 25 and the rlimit is 100, many of the specialization problems are being solved. However, when the rlimit is 100 and the number of training instances is 100, many of the specialization problems are not being fully solved. Many of the problems that were solved at 25 training instances are not solved at 100 training instances. Figure 5.7b provides evidence of this claims. At slightly less and greater rlimits, 75 and 125, average error rate tends to decrease monotonically with training set size.

The other two domains, auto diagnosis and nematode identification, also show that greater rlimits produce lower asymptotic error and error decreases monotonically with increased training set size. Figure 5.8 shows CV error as a function of training set size for the auto diagnosis and the nematode identification domains.

5.7 RIO Ablation Studies

In this section, results are presented that show the benefit of certain RIO techniques. All experiments were performed using the mutated rulebases used in the previous section. In order to determine the benefit of an RIO technique, sets of experiments were performed that omitted the use of individual RIO techniques.

A. Main results



B. Further evidence

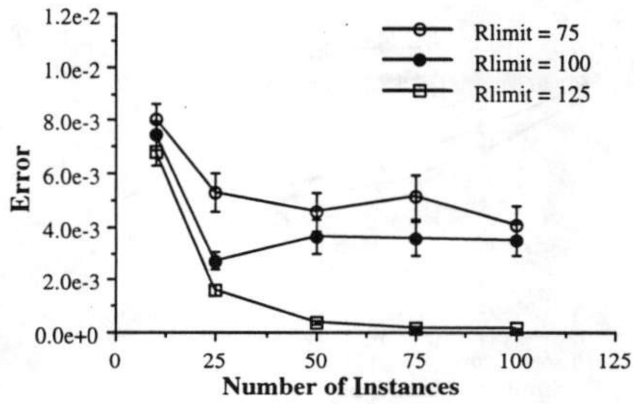
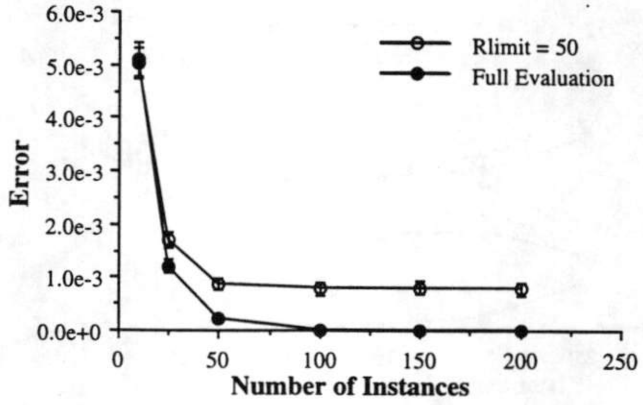


Figure 5.7: Average CV error as a function of training set size for various rlimit settings for the student loan domain. For full evaluation, at most 348 low-level revisions were evaluated.

A. Auto Diagnosis Domain



B. Nematode Identification Domain

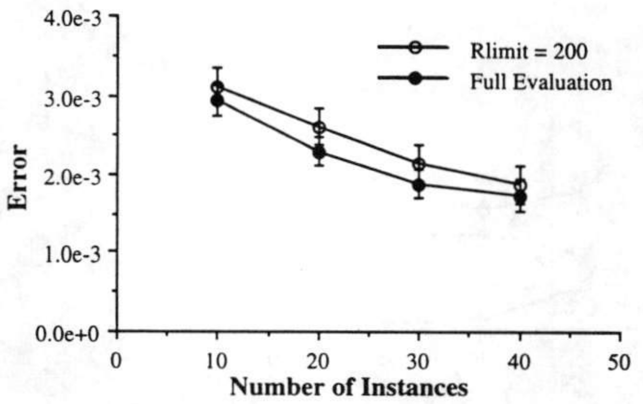


Figure 5.8: Average CV error as a function of training set size for various rlimit settings for the auto diagnosis and nematode identification domains. For full evaluation, at most 328 and 1404 low-level revisions were evaluated, respectively.

The first RIO technique to be considered is EXP. Based on a set of experiments where EXP was omitted, there was no benefit in using this technique on any of these domains when revisions were fully evaluated. This is not surprising given that no RIO distribution for any of the mutated student loan, auto diagnosis or nematode identification rules depended exclusively on EXP. In fact, none of the student loan and nematode identification mutated rulebases relied on EXP at all. For the auto diagnosis domain, all missing CE and increase salience mutated rulebases were able to use ASSUME in the absence of EXP.

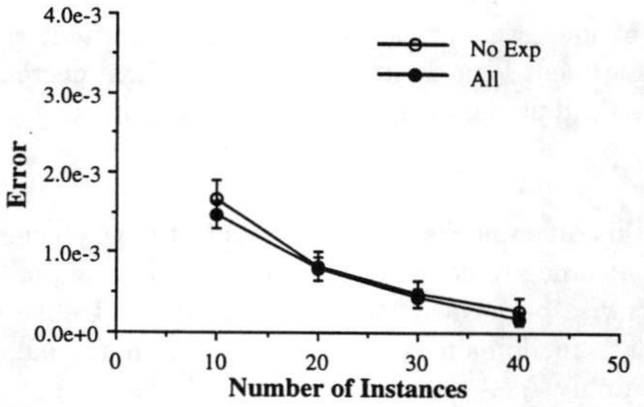
When the number of low-level revision evaluations is limited, the benefit of EXP is mixed. For the student loan domain, omission of EXP decreases error. While for the auto diagnosis and nematode identification experiments, the omission of EXP increases error.

An explanation for this difference is that EXP gets in the way when revising the student loan domain. It tends to corrupt the composite ordering of high-level revisions that would otherwise be formed mainly by ASSUME. Unlike the other two domains, the student loan domain is a very simple domain that is highly amenable to revision using only ASSUME and INCREASE. In fact, ASSUME is well suited to the identification and ordering of high-level revisions for the student loan domain because it is based on a blame assignment technique that requires that evidence of a rule's firing (not firing) be present in the final fact-list. Since all mutated student loan rulebases generate the deductive closure of the initial facts, ASSUME is always able to trace back from the final fact-list to determine which rule should or should not have fired. For the auto diagnosis and nematode identification domains, ASSUME is less well suited, and the strength of EXP's experimentation-based approach helps to place the correct revision higher in the ordering.

At this point it should be noted that none of the mutated rulebases for these three domains relied 100% on EXP because the benefit of EXP could not be effectively tested. One of the reasons that no nematode identification domain mutated rulebases was 100% dependent on EXP has to do with the retract-everything characteristic of this domain. The absence of overly general rules significantly reduces the potential for 100% EXP dependency.

In order to further test EXP using the nematode identification domain, a set of eight doubly mutated rulebases was formed that all included a missing positive pattern CE and a missing retract for the CE. Experiments were run with and without EXP using a version of SPR that, when adding a positive pattern CE to a rule, would also consider the addition of a retract for that CE. This modification gave SPR the potential to correctly revise the doubly mutated rulebases in a single execution.

A. Full Evaluation



B. Partial Evaluation

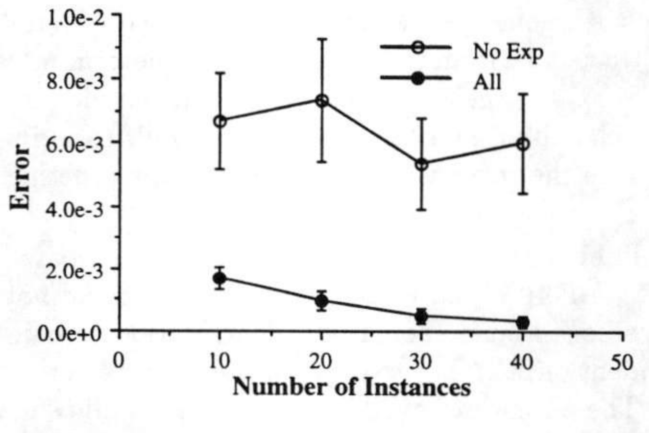


Figure 5.9: Average CV error as a function of training set size for the eight missing CE/retract mutated nematode identification rulebases when revised using a modified version of SPR.

A comparison showing the benefit of using Exp relative to the benefit of not using it, showed that at both full and partial evaluation, this slightly modified version of SPR performed better with Exp than without it. The difference was most pronounced at partial evaluation. Figure 5.9 shows these comparisons.

The second RIO technique being evaluated in this section is Misc. As with Exp, experiments were performed that omitted the use of Misc. Unlike with the omission of Exp, there is no benefit in using Misc for results that are based on the student loan and the auto diagnosis domains at either full or partial evaluation. In other words, Misc did not help or hurt for these two domains. For the student loan domain, no revision problems even required the use of Misc. For the auto diagnosis domain, some revision problems were able to use Misc, but no mutated auto diagnosis rulebase was 100% dependent on Misc.

For the nematode identification domain, the omission of Misc at full and partial evaluation showed that it was a benefit for reducing error (see Figure 5.10). This is not surprising since some of these mutated rulebases were 100% dependent on Misc.

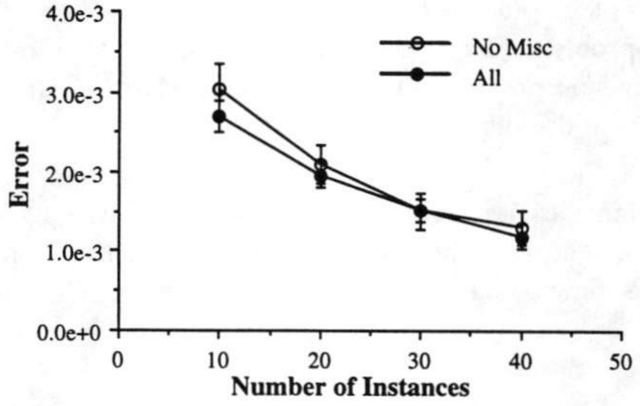
The other two RIO techniques, Assume and Increase, were not empirically validated. Based on the tree in Figure 4.2, the many partitions of the revision problem space that are dependent on Assume show its importance. If Increase is absent, no decrease rule mutated rulebases can be repaired.

5.8 High-Level Revision Ordering

Beyond the question of the benefit of each of the RIO techniques, is the question of how well the RIO techniques would do at ordering revisions. The results analyzed for this section are based on experiments where the composite ordering of high-level revisions produced by the RIO techniques was reversed prior to evaluation.

Figure 5.11 shows a comparison between using the normal ordering and the reversed ordering for the student loan, auto diagnosis and nematode identification domains. For each graph, error is presented as a function of rlimit. Based on these graphs, it is clear that reversing the ordering produces significantly worse results.

A. Full Evaluation



B. Partial Evaluation

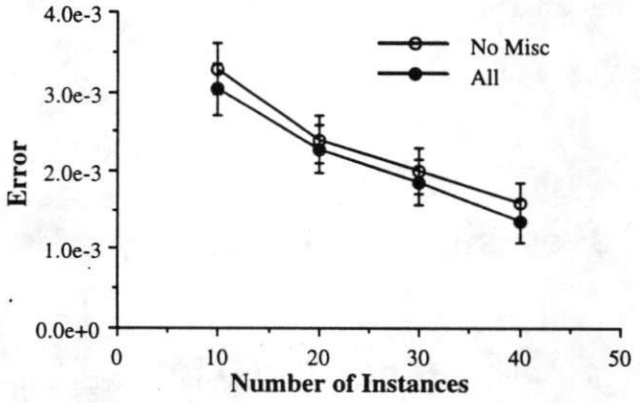
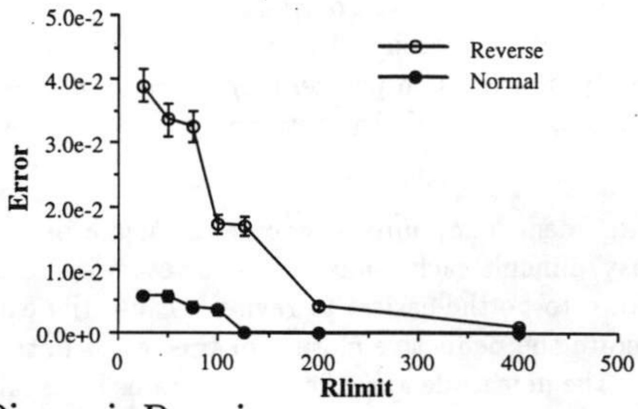
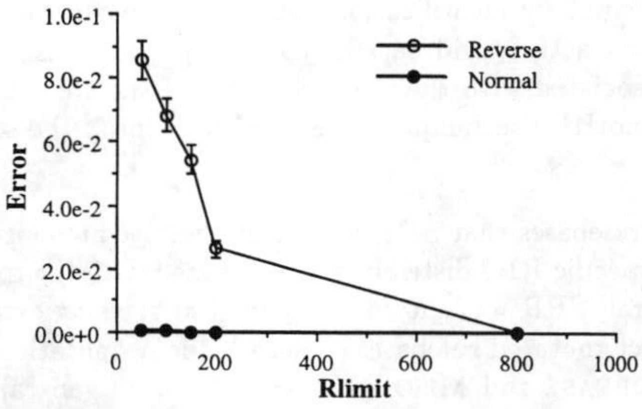


Figure 5.10: Average CV error as a function of training set size for the singly mutated nematode identification rulebases.

A. Student Loan Domain



B. Auto Diagnosis Domain



C. Nematode Identification Domain

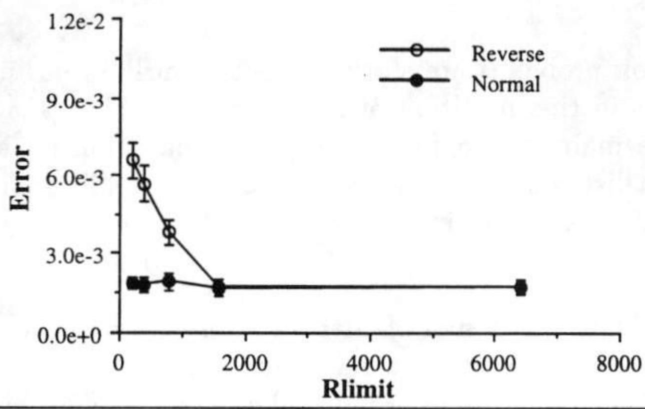


Figure 5.11: Average CV error as a function of rlimit for the reversed and normal ordering of high-level revisions.

5.9 Chapter Summary

This chapter presented an empirical analysis of SPR in the context of revising the types of revision problems that it is designed to revise. SPR, not CR2, is studied in this chapter. The types of revision problems that are used to evaluate SPR are singleton problems consistent with the revision problem space model described in Chapter 3.

For the three domains, student loan, auto diagnosis and nematode identification, it was shown how easy/difficult each domain was to revise by SPR. The student loan domain was shown to be the easiest to revise because the execution of its rulebase tends to generate the deductive closure of the set of initial facts. Neither the auto diagnosis nor the nematode identification domains have this characteristic. The auto diagnosis domain was next in difficulty because it has rules with different saliences and because the execution of a rule depends on the order of rule execution. Lastly, the nematode identification domain was shown to be most difficult because it uses retract actions and because some singleton revision problems for this domain are associated with the uncovered leaf of the tree shown in Figure 4.2. In other words, no RIO technique is designed to identify the solution to these revision problems.

The types of mutated rulebases that SPR was most effective at revising depended significantly on the specific RIO distributions associated with the mutated rulebases. However, in general, SPR was shown to be good at revising extra CE, extra assert and extra retract mutated rulebases. These kinds of mutations were always identified by the INCREASE and MISC RIO techniques. In general, SPR found it most difficult to revise missing CE revisions when training set size was small. At low training set sizes, many add CE revisions tend to produce the same evaluation over the instances.

Since uncovered revision problems are shown to occur fairly often, not being able to identify revisions in this partition of the revision problem space is a significant problem. As the main reason for an uncovered partition is the sole reliance of SPR on final fact-list constraints, it would seem that additional forms of constraints will be needed in order to, in general, solve the production system revision problem.

In order to further evaluate the design of SPR, an ablation study was conducted that showed when each RIO technique was important toward revising singly mutated rulebases. The composite ordering of high-level revisions was also studied. With respect to placing a resource limit on the number of low-level revisions evaluated by SPR, limiting this resource tended to increase asymptotic error as training set size increased.

Chapter 6
CR2: Results & Evaluation

6.1 Chapter Overview

CR2 is a hill-climbing revision system that uses SPR to revise rulebases with multiple problems. SPR is designed in the context of a set of explicit assumptions. When used by CR2, these assumptions are regularly violated. An important goal of this chapter is to determine empirically whether the violated assumptions significantly affect CR2's ability at revising rulebases with multiple problems.

All results in this chapter are based on the use of CR2, not SPR, as the revision system. Each experiment revises a mutated rulebase that is two or more revisions away from being correct. All mutated rulebases have the potential of being correctly revised by CR2.

6.2 Experimental Methodology

In this chapter, experiments were performed using the three domains: student loan, auto diagnosis and nematode identification. The experiments performed for this section were performed using mutated rulebases that have multiple mutations. For each domain, three sets of 50 mutated rulebases were generated that included two, four and six mutations. Mutated rulebases were organized by the number of mutations made to the rulebases. All mutated rulebases were inconsistent with some constraints.

The averages corresponding to the data points used to form graphs were computed using the results of at least 20 individual experiments. Confidence intervals are at two standard deviations (95.44%).

Table 6.1: Average initial error for each domain's mutated rulebase sets evaluated over all instances. There are three sets of mutated rulebases (three mutation levels) for each domain.

Domains (# Muts)	**CV Error mean (sd)**	**Elimit Error mean (sd)**
Student Loan (2)	6.83e-2 (6.77e-2)	0.00e-0 (0.00e-0)
Student Loan (4)	1.34e-1 (8.86e-2)	0.00e-0 (0.00e-0)
Student Loan (6)	1.70e-1 (8.52e-2)	0.00e-0 (0.00e-0)
Auto Diagnosis (2)	1.02e-1 (1.43e-1)	0.00e-0 (0.00e-0)
Auto Diagnosis (4)	1.76e-1 (1.60e-1)	0.00e-0 (0.00e-0)
Auto Diagnosis (6)	1.81e-1 (1.73e-1)	0.00e-0 (0.00e-0)
Nematode Identification (2)	1.14e-2 (1.47e-2)	1.09e-1 (2.41e-1)
Nematode Identification (4)	1.40e-2 (1.54e-2)	1.10e-1 (2.54e-1)
Nematode Identification (6)	2.13e-2 (1.77e-2)	8.69e-2 (2.27e-1)

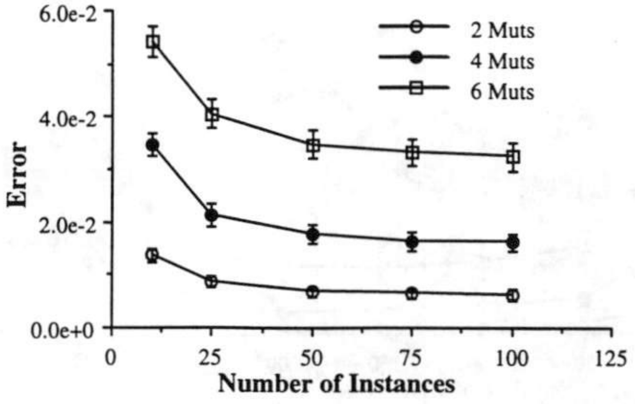
The mean and standard deviation of the CV and elimit errors across the mutated rulebases for each domain are presented in Table 6.1. Only the mutated nematode identification rulebases produce elimit errors.

6.3 Results & Analysis

The results in this section show how rlimit, training set size, and the number of mutations made to a rulebase affect CV error. The graphs in Figure 6.1, show CV error as a function of training set size for the student loan domain. Each graph shows learning curves for the three sets of mutated student loan rulebases. The top graph, Figure 6.1a, was constructed using the results from experiments where the number of low-level revision evaluations (rlimit) was limited to 25. The curves in this graph shows that as training set size increases, error decreases. In addition, the graph shows that less mutated rulebases tend to produce lower after-revision errors.

The bottom graph in Figure 6.1 was constructed using the results of similar experiments, except that no limit was imposed on the number of low-level revision evaluations. This graph shows qualitatively similar curves to the curves in the top graph except that the average error after revision tends to be lower at full evaluation. This result is consistent with the results that compared full and partial evaluation in Chapter 5.

A. Rlimit = 25



B. Full Evaluation

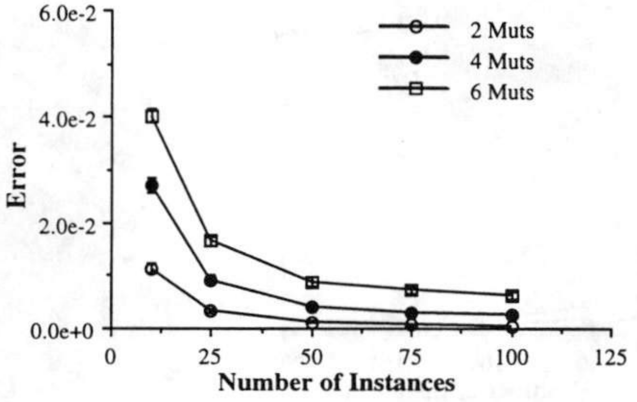
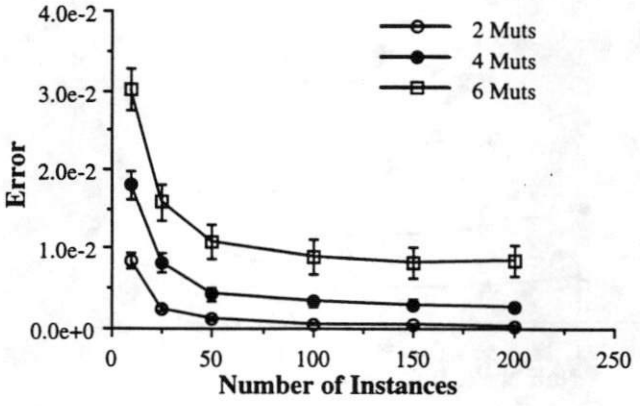


Figure 6.1: Average CV error as a function of the number of training instances for the student loan domain. For full evaluation of the six mutation rulebases, at most 914 low-level revisions were identified and evaluated.

A. Rlimit = 50



B. Full Evaluation

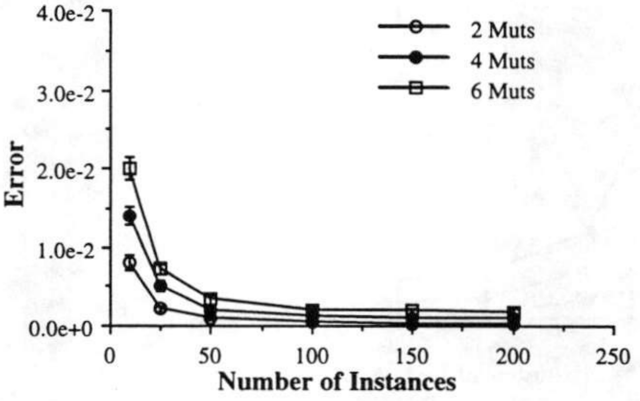
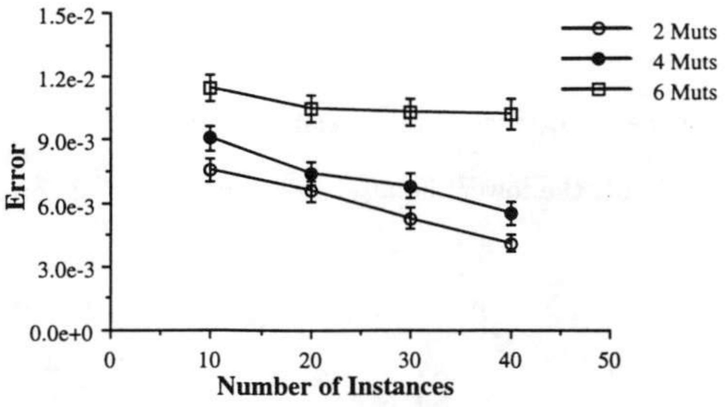


Figure 6.2: Average CV error as a function of the number of training instances for the auto diagnosis domain. For full evaluation of the six mutation rulebases, at most 577 low-level revisions were identified and evaluated.

A. Rlimit = 50



B. Full Evaluation

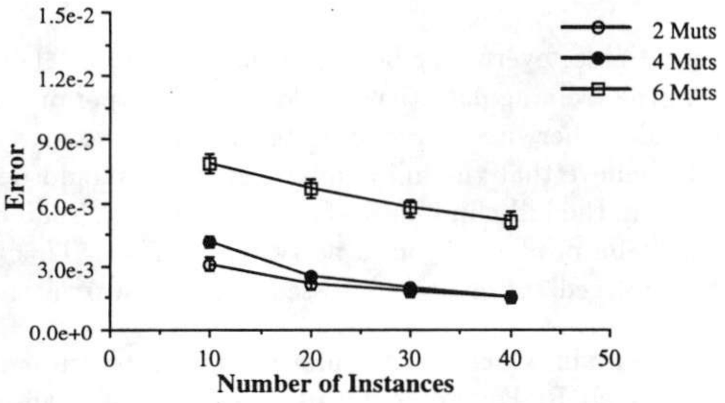


Figure 6.3: Average CV error as a function of the number of training instances for the nematode identification domain. For full evaluation of the six mutation rulebases, at most 2819 low-level revisions were identified and evaluated.

Figures 6.2 and 6.3 show graphs for the auto diagnosis and nematode identification domains. These graphs were constructed using the results of the same types of experiments that were used to construct the graphs in Figure 6.1. For all domains,

- As training set size increases, error decreases.
- Less mutated rulebases produce lower after-revision asymptotic error.
- The greater the rlimit, the lower the after-revision asymptotic error.

6.4 Overfitting

A significant concern for most hill-climbing optimization systems, like CR2, is the problem of overfitting the training instances at the expense of reducing consistency with unseen instances. Excessive optimization of the system to be more accurate on training instances may cause the system to stray from a simpler, more general, and accurate overall solution. While it is true that CR2 has the potential for this problem, it is important to note how and where it faces this problem.

CR2 has the potential for overfitting because it has the potential of following spurious correlations in the training data that would cause it to perform more hill-climbing steps than would otherwise be appropriate. In other words, since there is no a priori reason to believe that the hill-climbing gradient should be uniform, appropriately slight rises in the hill-climbing surface are indistinguishable from the spurious rises that might be produced from a noisy training set. This particular problem has not been observed and is not addressed in this research.

Another perceived way in which CR2 could face this problem is in the use of too large of a rlimit by SPR. Fortunately, this is not in reality an overfitting problem. SPR is designed, using the RIO techniques, to generate an ordering of revisions. The ordering of revisions is produced using information available in the training instances and is designed to be an approximation of the ordering that would be produced if each revision was fully evaluated over all of the instances. An increase in the rlimit will cause revisions with lower potential to be evaluated and considered during each hill-climbing step. Evaluation of more revisions will not produce a more complex rulebase. The evaluation of more revisions makes it more likely that the correct revision is identified. The decision of what rlimit to use is a resource limitation issue.

6.5 Chapter Summary

This chapter presented an empirical analysis of CR2 in the context of revising rulebases with multiple mutations. The main goal of this chapter was to determine whether SPR could be used by CR2 to revise rulebases that are not consistent with the assumptions under which it was designed. To this end, the results clearly show that SPR is able to revise rulebases that have multiple problems.

The results also showed that the effect of changing training set size and rlimit are the same for CR2, as they are for SPR (as described in Chapter 5). Specifically, as training set size increases, error decreases and the greater the rlimit, the lower the after-revision asymptotic error. Unfortunately, as the number of mutations increases, the average asymptotic error also increases. With increasing numbers of mutations to identify and repair, SPR appears to get lost, even for the student loan domain. Interactions among the individual revision problems (mutations) caused SPR to not find the correct revisions and to select incorrect revisions. This is a problem that is classic to many theory revisions systems, e.g. A3 and FORTE.

Chapter 7
Rule Structure Filtering

7.1 Chapter Overview

The last step prior to the selection of a best revision by SPR is rule structure filtering. The goal of rule structure filtering is to filter the set of equally evaluated revisions so that revisions that produce ill-structured rules are avoided. The main reason for rule structure filtering is to improve accuracy. A secondary reason is to enhance the understandability by enhancing the organization of the rulebase.

This chapter presents a description of issues concerning rule and rulebase structure, an approach to rule structure filtering, and an empirical analysis of when rulebase structure filtering works to improve accuracy. Examples will demonstrate how rule structure filtering can maintain or enhance both organization and understandability of the rulebase.

7.2 Experimental Methodology

In this chapter, experiments were performed using the three domains: student loan, auto diagnosis and nematode identification. The experiments performed were completed using the mutated rulebases constructed for Chapters 5 and 6. Some of the results are based on the use of SPR and some are based on the use of CR2 as the reviser.

The empirical analyses in this chapter are based on comparisons between using and not using rule structure filtering. All results presented in previous chapters were based on the use of rule structure filtering.

The averages corresponding to the data points used to form graphs were computed using the results of at least 20 individual experiments. Confidence intervals are at two standard deviations (95.44%).

7.3 Rule & Rulebase Structure

It has long been recognized that the structure of a rulebase and its rules are an important consideration when trying to understand a rulebase. For example, rule groups are often used to reduce the complexity of the rulebase so that it may be described in terms of rule group abstractions instead of individual rules.

Groups of rules may be associated with particular purposes, and may be used to understand the flow of rule execution. Some researchers argue that rule grouping may be an approach toward verification (Mehrotra, 1991) and maintenance (Jacob & Froscher, 1990) of rulebases. Rule groups can also be looked at as routines, subroutines and functions in a procedural programming language.

Groups of rules themselves may have internal structure. That is, individual rule groups may be best understood in terms of rule subgroups. For example, recursive Horn clause predicates are composed of multiple rules (or clauses) and have structure. In order for them to terminate, recursive Horn clause predicates tend to be composed of two groups of clauses. They usually have one or more non-recursive base-case clauses followed by recursive clauses. The `member` predicate, for example, is defined as:

```
member(H, [H|T]).
member(H, [X|T]) :- member(H,T).
```

Together, these two clauses form a rule group, and separately, they form two rule subgroups (of one rule each). The first clause (rule subgroup) is a base-case for defining list membership. It has the interpretation: an element H is a member of a list $[H|T]$ if it is the head of the list. The second clause is recursive and has the interpretation: an element H is a member of a list if it is a member of the tail T of the list.

The auto diagnosis and nematode identification domains have strong rule group structure. Figures 7.1 and 7.2 (repeated in Chapter 2) show rule groupings and flow-of-control between rule groups for the auto diagnosis and nematode identification domains, respectively. For these domains, rule groups were formed manually and are based on a subjective interpretation by this author. Existing research, to automatically form rule groups from CLIPS rulebases includes: (Jacob & Froscher, 1990), (Anastasiadis, 1991) and (Grossner, Preece, Chander, Radhakrishnan & Suen, 1993).

In addition to being grouped by purpose, rules may also be grouped by rule structure similarity. For the rulebases used in this research, rules with a common purpose tend to have one of a small number of common structures. A

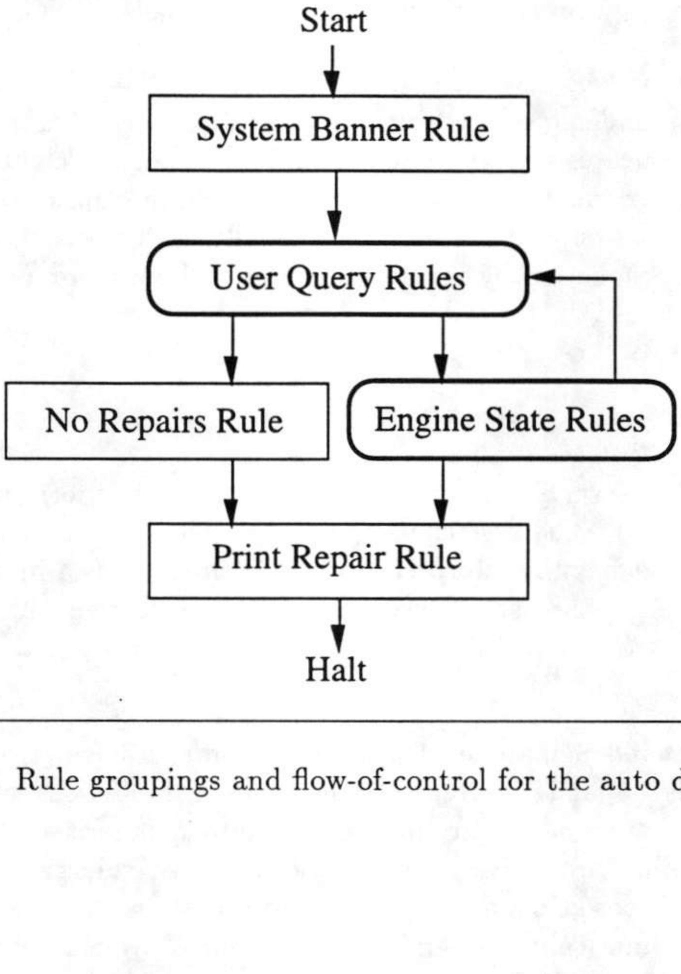


Figure 7.1. Rule groupings and flow-of-control for the auto diagnosis rulebase.

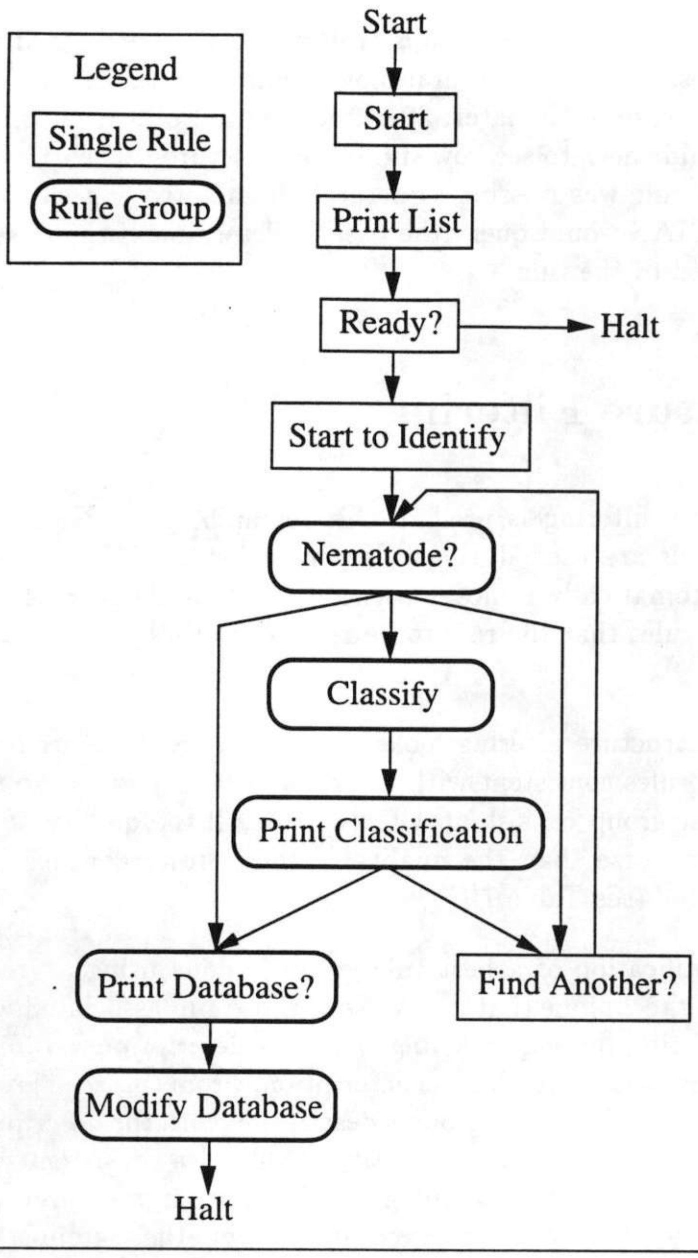


Figure 7.2: Rule groupings and flow-of-control for the nematode identification rulebase. This is a duplicate of Figure 2.2.

classic example of previous research that took advantage of rule grouping by rule similarity was the use of rule models in TEIRESIAS (Davis, 1979). TEIRESIAS is a knowledge acquisition system that was designed to generate rulebases similar in format to the MYCIN (Buchanan & Shortliffe, 1984) rulebase.

In TEIRESIAS, hierarchies of rule groups (rule models) were formed and used for a number of purposes including natural language interpretation and rule generation checking. In the case of the later, after the creation of a new rule by the user, TEIRESIAS would check to see how similar in structure it is with other rules. If the newly created rule was missing structural characteristics that other similar rules had, TEIRESIAS would query the user to determine if the missing components should be added to the rule.

7.4 Rule Structure Filtering

The way rule structure filtering is used in this research is similar to one of the ways that rule models are used during rule generation in TEIRESIAS. Rule structure filtering automatically removes revisions that would increase the complexity of the group of rules that the rule to be revised would belong to after being revised.

In other words, rule structure filtering looks at the rule being revised and identifies the best group of rules consistent with the rule. It then revises the rule and again identifies the best group consistent with the rule. If the quality of the revised rule's best group is worse than the quality of the unrevised rule's best group, the revision is removed (see Table 7.1).

In this research, identification of a best rule group is done using a greedy agglomerative-like clustering technique that uses as seed the group associated with the description of the rule being revised. Starting with this description, an initial group is formed from all rules consistent with the description. From this seed group, other groups are formed by generalizing the group's description with the description of each rule not already in the group. After identifying all rules consistent with each new group's description, evaluation of the groups' quality is measured and compared to the quality of the best groups observed so far. From these groups, one of the highest quality groups is selected as the seed group for the next iteration. After a fixed number of iterations (10 iterations for this research), the best quality observed is returned as the quality of the best group for the rule being revised.

The structure of rules and groups of rules are described in terms of a set of feature-value pairs. All individual rules are described using the same number of

Table 7.1. High-level description of the rule structure filtering algorithm.

```
rule_structure_filtering(rulebase, revisions)
  {
    original_revisions = revisions
    for each revision in revisions
      unrevised_rule = revision_rule(revision)
      before_group = find_best_group(rulebase, unrevised_rule)
      revised_rule = revise_rule(rule, revision)
      after_group = find_best_group(rulebase, revised_rule)
      if quality(before_group) > quality(after_group)
        remove revision from revisions
    if revisions is empty
      then return original_revisions
      else return revisions
  }
```

pairs. A rule belongs to a group of rules if its feature-value pairs are a superset of the feature-value pairs that describe the group. The description of a group of rules is equivalent to the intersection of the descriptions of the individual rules in the group. The features used to describe rules and groups of rules are listed in Table 7.2.

These features are divided into three classes. The first class contains features that describe how many of each type of component a rule has. For example, *CE-Count* is a number that is set to the number of CEs that a rule has. *Positive-Pattern-CE-Count* is the number of positive pattern CEs in a rule and is less than or equal to *CE-Count.* Figure 7.3 shows dependencies between some of these features. The second class contains features that are abstractions of the first class. For example, *CE-Count-p* has value *some* when *CE-Count* is greater than 0, otherwise it has value *none.* The last class consists only of the features *Salience* and *Retracts-All. Salience* is simply the salience of the rule. *Retracts-All* is a boolean feature that has value *yes* if there exists a retract for each positive pattern CE in the rule.

The quality of a group of rule is a function of the number of rules in the group and the number of features used to describe the group. The larger the number of rules and size of the description, the better the quality of the group. The expression used to calculate group quality is $Description\_Size + \log_2(Group\_Size)$, where $Description\_Size$ is the number of feature-value pairs that describe the group and $Group\_Size$ is the number of rules in the group. Two groups have the same quality

Table 7.2: Features and possible values used to describe individual rules and rule groups. Some features are functions or pure abstractions of other features.

Features	Possible Values
ce-count	integer
pattern-ce-count	integer
pos-pattern-ce-count	integer
neg-pattern-ce-count	integer
test-ce-count	integer
pos-test-ce-count	integer
neg-test-ce-count	integer
action-count	integer
assert-count	integer
retract-count	integer
printout-count	integer
bind-count	integer
ce-count-p	some, none
pattern-ce-count-p	some, none
pos-pattern-ce-count-p	some, none
neg-pattern-ce-count-p	some, none
test-ce-count-p	some, none
pos-test-ce-count-p	some, none
neg-test-ce-count-p	some, none
action-count-p	some, none
assert-count-p	some, none
retract-count-p	some, none
printout-count-p	some, none
bind-count-p	some, none
salience	integer
retracts-all	yes, no

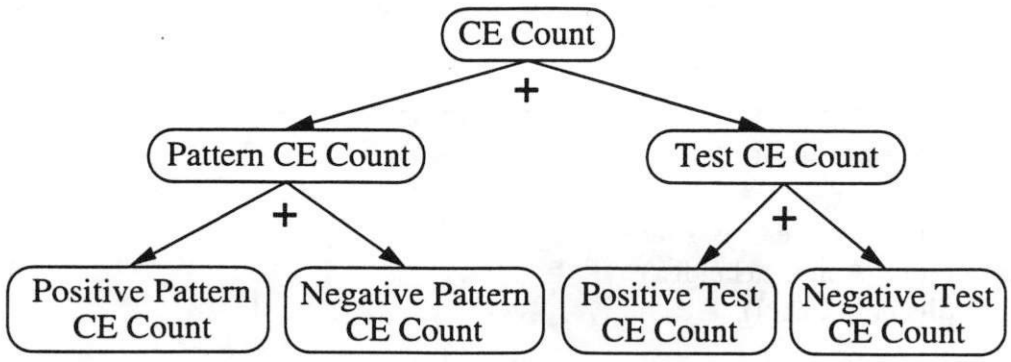


Figure 7.3: Example dependencies between some of the rule/rule group description features. CE-Count is the sum of Pattern-CE-Count and Test-CE-Count.

if a linear decrease in the number of feature-values is offset by an exponential increase in the number of rules covered by the group.

The particular feature set used here was chosen to enable rule grouping based on varying levels of structural similarity. Groups of rules that have members that differ by the specific types of CE counts, for example, may still have in common a common number of CEs overall. As rule group quality is based on the size of the rule group description, this choice of feature set allows for finer differences in rule group quality and a great variety of potential rule groups. In other words, had abstractions of features, e.g. *Positive-Pattern-CE-Count-p* and features that are based on the sum of the counts of other features, e.g. *CE-Count*, been omitted, most rule groups would either have very long descriptions or very short descriptions. The size of the rule groups with very long descriptions would tend to be small and the size of short description rule groups would tend to be large.

7.5 Results & Analysis

For this research, there are two main reasons for rule structure filtering. The first reason has to do with guiding the revision process to decrease error. The second reason relates to revision understandability and rulebase organization.

With respect to error reduction, when the rulebase has a small number of groups of similarly structured rules or when no revisions are needed to produce the occasional unique rule, rule structure filtering tends to work because it removes

Table 7.3: Example of how a revision could produce an ill-structured rule. This revision is effectively achieving the same result as the more appropriate delete rule revision.

A. Rule before revision

```
(defrule Aphelenchoidea-tail-shape
   (Superfamily Aphelenchoidea)
   (not (tail-shape ?size))
   =>
   (bind ?response (ask-question shape-tail?))
   (assert (tail-shape ?response)))
```

B. Rule after revision

```
(defrule Aphelenchoidea-tail-shape
   (Superfamily Aphelenchoidea)
   (not (tail-shape ?size))
   =>
   (bind ?response (ask-question shape-tail?))
```

spurious revisions that produce ill-structured rules. In the following sections, it will be shown, via empirical results, that rule structure filtering reduces error for the student loan, auto diagnosis and nematode identification domains.

Rule structure filtering also enhances revision understandability by helping SPR generate revisions in the same manner as a human rulebase reviser. For example, when a human reviser considers the revision from rule A (Table 7.3) to rule B, he (she) is likely to see the revision as producing an ill-structured rule. After all, all other rules with bind actions in the rulebase have assert actions that use the variable assigned by the bind. For this example, the revision is equivalent to rule deletion but is less understandable than the revision to delete the rule.

In terms of rulebase organization, human rulebase designers tend to follow personal organizationally themes that enhance understandability. For example, a designer may decided to design a rulebase using a single salience for all rules, even though certain subtasks may be accomplished using multiple saliences. The designer reasons that the simpler organization is important for ease of understanding. Rule structure filtering biases SPR toward revisions that are consistent with the initial organization of the rulebase. For this example, a revision that changes the salience of an existing rule is likely to be removed because it produces a rule that is different in structure from other rules.

Note, for the experienced human rulebase reviser, the observation that a revision produces an ill-structured rule may be used as a signal to attain a better understanding of the revision. Sometimes it is appropriate to generate a rule with a unique structure.

7.5.1 Singleton Revision Problems

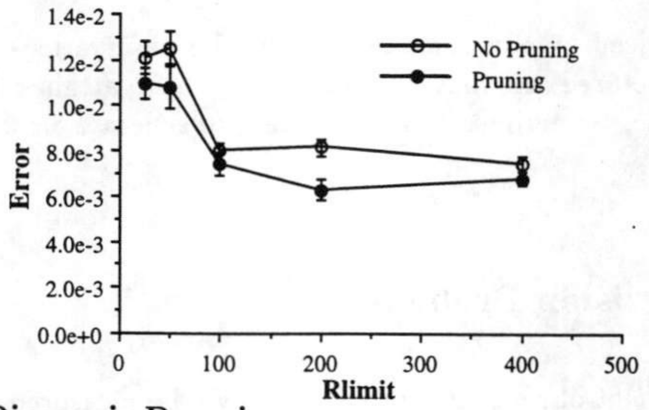
In this section, the benefit of rule structure filtering will be measured in the context of singleton revision problems. Results generated for this section were compared with results generated for Chapter 5. The reviser being used in this comparison is SPR.

For the first result, the bias of the rule structure filtering is evaluated in the context of the singleton mutated rulebases generated for each domain. For each mutated rulebase, an experiment was conducted to determined if the correct revision to the mutated rule would be pruned by rule structure filtering. For the student loan and nematode identification domains, 20% of the mutated rulebases were inconsistent with the rule structure filtering bias. For the auto diagnosis domain, only 16% of the mutated rulebases were inconsistent with this bias.

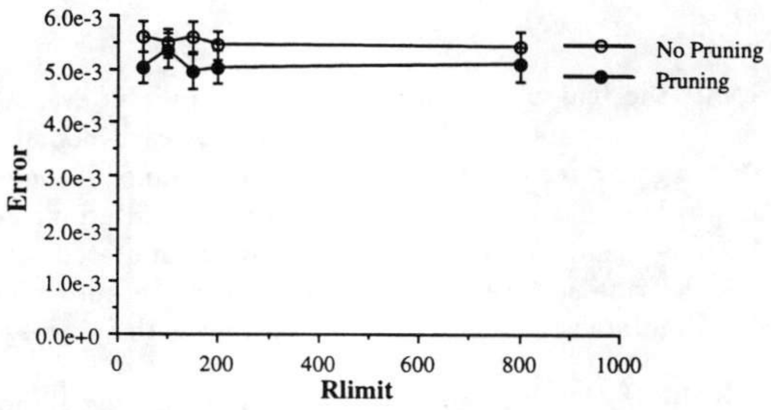
The benefit of the filtering bias was not distributed evenly among all problem types. For the student loan domain, filtering was inappropriate when the mutation to the rulebase was a deleted test CE. Most of the rules in the rulebase have no test CEs, so the correct addition of a test CE to the revised rule was seen as an ill-structured revision. For the auto diagnosis domain, filtering was inappropriate when the mutation to the rulebase was the deletion of an assert from a two assert rule. Most of the rules in these rulebases have only a single assert, so the revision of adding an assert caused the revised rule to become dissimilar from most of the other rules.

For the next set of results, the benefit of rule structure filtering at reducing error is measured. The graphs in Figure 7.4 demonstrate that the lack of filtering increases error for each of the three domains at 10 training instances. At 25 training instances, for the student loan and auto diagnosis domains, the difference in error is much smaller (see Figure 7.5). The same comparisons at higher training sets sizes show that the difference between using and not using rule structure filtering is indistinguishable.

A. Student Loan Domain



B. Auto Diagnosis Domain



C. Nematode Identification Domain

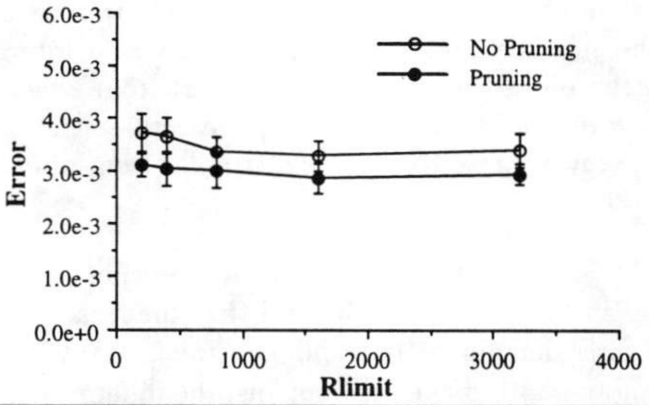
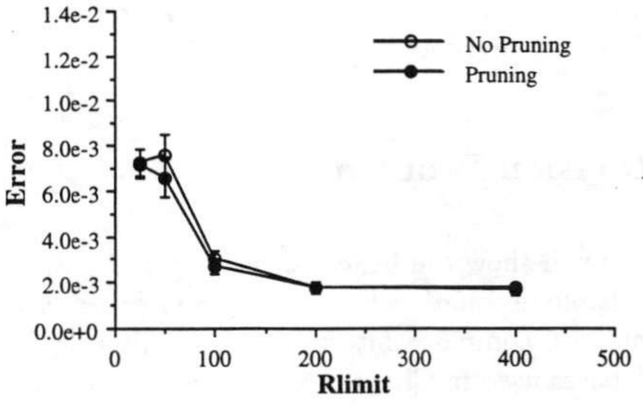


Figure 7.4: A comparison of CV error when using and not using rule structure filtering. Results are based on the use of SPR and a training set size of 10.

A. Student Loan Domain



B. Auto Diagnosis Domain

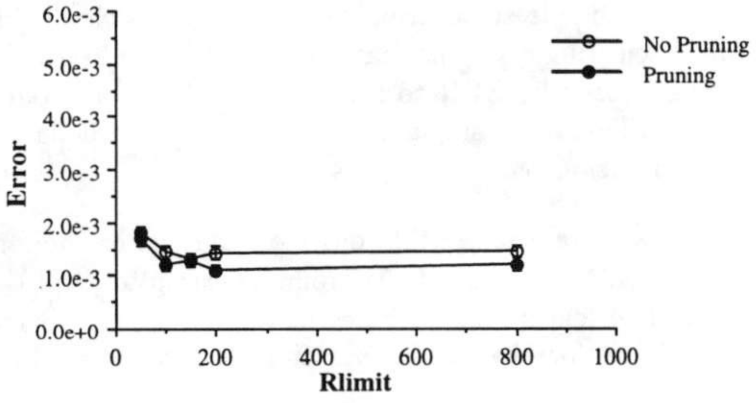


Figure 7.5: A comparison of CV error when using and not using rule structure filtering. These results are based on the use of SPR, and a training set size of 25 instances.

The indistinguishability of results at higher training set sizes can be explained by recognizing that at lower training set sizes, more revisions end up with the same best evaluation. Evaluation over the instances produces less discrimination at small training set sizes. When filtering is appropriate for a mutated rulebase, filtering will have the affect of removing spurious revisions and thus increasing the chance of randomly selecting the correct revision.

7.5.2 Multiple Revision Problems

In this section, results will show the benefit of using filtering when the rulebase being revised has multiple revision problems. The reviser being used in this comparison is CR2. Figure 7.6 compares filtering with no filtering for the three domains and mutated rulebases used in Chapter 6. Each of the mutated rulebases had six mutations and all selected revisions were evaluated. As with singleton revision problems and SPR, for each of the three domains, the difference in error is again greatest at lower training set sizes.

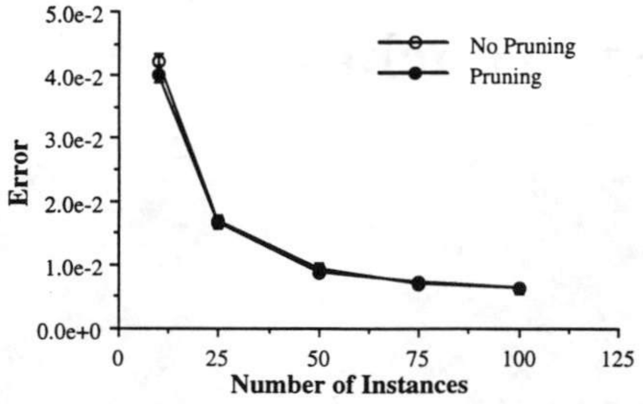
7.6 Chapter Summary

This chapter presented rule structure filtering, an approach that biases the revision system toward selecting revisions that do not produce ill-structured rules. Rule structure filtering is used by SPR to filter the revision that would produce ill-structured rules from the set of best evaluated revisions. It is the last step prior to selection of the best revision.

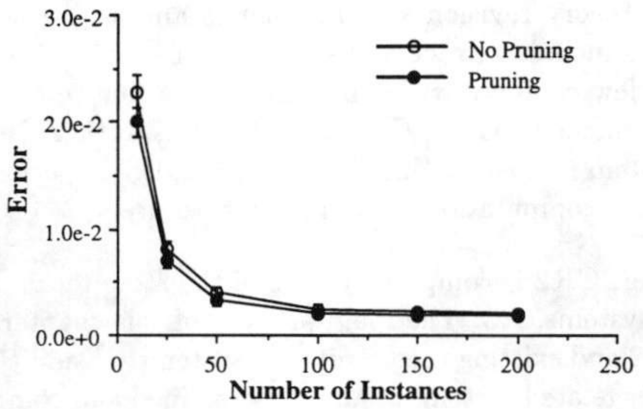
The approach uses a set of features that describe rule groups and a rule group quality metric that is based on the size of the group's description and the number of rules in the group. The feature set includes features that are abstractions of other features which allows for the characterization of a range of differing rule groups.

Empirical results were presented that showed the benefit of rule structure filtering at decreasing error when training set sizes were small. When training set sizes were larger, the results of using and not using rule structure filtering were indistinguishable because evaluation over the training instances produce smaller numbers of best evaluated revisions to filter. At lower training set sizes, rule structure filtering was able to remove revisions that were spuriously evaluated as best.

A. Student Loan Domain



B. Auto Diagnosis Domain



C. Nematode Identification Domain

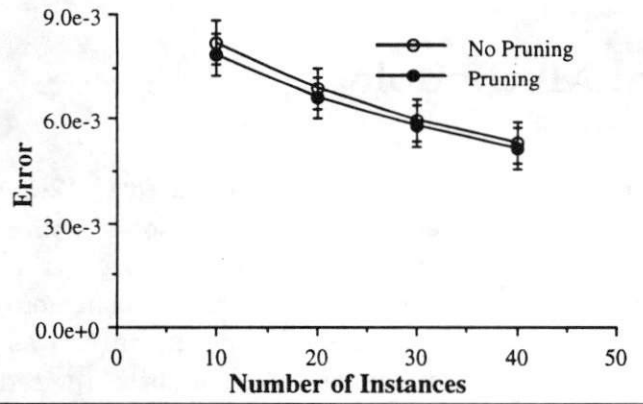


Figure 7.6: A comparison of CV error when using and not using rule structure filtering. Results are based on the use of CR2 over mutated rulebases that each have six mutations.

Chapter 8
A Comparison to Other Approaches

8.1 Chapter Overview

CR2 is one of the few theory revision systems that is able to revise production system rulebases that include retract and other actions, as well as rule saliences. As such, there are few competitors in the field with which to compare and contrast CR2. The forerunner to CR2, CLIPS-R (by this author), is also able to handle production system rulebases. Unfortunately, CLIPS-R performs a much less directed search and is computationally impractical to use.

Therefore, in this chapter, CR2 is compared to one of the more recent Horn clause-based theory revision systems, A3. The main purpose of this chapter is to show similarities between CR2 and existing theory revision systems. In addition to A3, CLIPS-R and three other related systems are briefly described and compared with CR2.

8.2 Experimental Methodology

For this chapter, experiments were performed that compared CR2 and another theory revision algorithm, A3. Experiments were run on a total of three mutated rulebases from two domains. Experiments consisted of forming a train/test partition of the instances and then revising a mutated rulebase using only the training instances (train partition). CV error on the training instances was measured after revision. Experiments varied by the mutated rulebase revised, the training set size, rlimit (for CR2), and the specific train/test partition of the instances. Unlike for previous results, the number of intermediate concepts (other constraints) available to guide revision was also varied.

The averages corresponding to the data points used to form graphs were computed using the results of at least 40 individual experiments. Confidence intervals are at two standard deviations (95.44%). Only the results from CR2 have confidence intervals.

8.3 Domains

The two domains tested in this chapter are the student loan domain and the king-rook-king domain. Both domains have rulebases that were originally encoded as Horn clauses.

Two mutated versions of the student loan rulebase were used for this chapter. One rulebase had four mutations which included a missing CE, missing rule, an extra CE and an extra rule. This mutated version of the student loan domain has been used in previous research to demonstrate how well algorithms do at revising rulebases with a variety of problems (Pazzani & Brunk, 1991). The second mutated student loan rulebase is identical to the first, except that it does not have the missing rule mutation. See Appendix A for a description of the four-mutation version of the rulebase.

For the results of this chapter, 100 instances were randomly selected from the original 1000 instances used in previous chapters. As before, each instance included the target concept and eight intermediate concepts as potential constraints.

A Horn clause version of the king-rook-king domain has been used by many researchers (Muggleton, Bain, Hayes-Michie & Michie, 1989; Muggleton & Feng, 1990; Pazzani & Kibler, 1992; Richards, 1992). The main task of the king-rook-king domain is to decide which chess board positions containing a white king, white rook and black king are illegal. A set of positions is considered illegal if the black king is in check or if any two pieces occupy the same board position. This domain is also sometimes referred to as "illegal".

The original Horn clause version of the king-rook-king rulebase could not be converted directly into a production system rulebase because many of the clauses in the Horn clause version had variables in their heads that were not used and not bound in the body of the clause. A production rule formed from such a clause would have an assert for the head of the clause where not all variables are bound. Such rules are illegal for the production system language used in this research. Therefore, a new production system version of the king-rook-king rulebase was produced from scratch. The rulebase was produced in such a way that it encoded the same task as the original Horn clause version of the king-rook-king domain and

Table 8.1: Initial errors for each of the mutated rulebases evaluated over all instances.

Domains	CV Error Target Only	CV Error All Constraints
Student Loan (4 muts)	2.50e-1	1.03e-1
Student Loan (3 muts)	2.40e-1	9.56e-2
King-Rook-King	2.50e-1	1.65e-1

also was easily encoded as a Horn clause rulebase. The new version had 15 rules and used the same background information as the original Horn clause version.

A single mutated version of the king-rook-king domain was generated. This mutated rulebase had three missing CE mutations and two extra CE mutations in five rules. 200 instances were randomly generated from the correct king-rook-king rulebase. In addition to the target concept, each instance included three intermediate concepts. See Appendix D for a more detailed description of the king-rook-king domain.

Table 8.1 presents the initial error when evaluated over all instances for the target concept and for all constraints (target and intermediates) for each of the mutated rulebases.

8.4 CR2 versus A3

This section provides a description of the theory revision system A3, as well as an analytical and empirical comparison of the two systems.

The main reason A3 was chosen for comparison to CR2 is because it is one of the newest and most flexible available theory revision systems. Like most systems, A3 was designed to revise Horn clause rulebases. Specifically, A3 is able to revise relational Horn clause theories that include negated literals. Also A3 is able to take advantage of constraints on what should and should not be provable by any predicate in the theory.

Unfortunately, the selection of A3 for comparison to CR2 is less than ideal. It would have been better to have chosen a system that is able to revise production system rulebases that are similar to the rulebases revised by CR2. Alas, few such available systems exist. Therefore comparisons in this section and the next will

Table 8.2. High-level description of A3.

```
A3(theory, examples)
  {
    while theory accuracy over examples is increasing do
      assumptions = find_best_assumptions(theory, examples)
      theory = modify_theory(assumptions, theory, examples)
    return theory
  }
```

focus on abilities related to the revision of rulebases that can readily be converted to Horn clause rulebases. In other words, little will be said regarding rule salience, retract actions and multiple actions in a rule.

8.4.1 Analytical Comparison

A3, like CR2, is a true theory revision system in that it is able to directly revise and return the input theory. Like CR2, A3 has a set of revision operators that generate revisions that are each evaluated over the instances. A3 works as a hill-climbing system that identifies the best revision to the system, performs that revision and then repeats. When further revisions cannot improve the theory, the current best theory is returned. Table 8.2 shows a high-level description of A3.

The information used by A3 and CR2 to constrain the revision process differs somewhat. A3 uses examples similar to CR2's constraints. Examples describe what should and should not be provable by the theory, and may correspond to any predicate in the theory. CR2 has to notion of an instance which includes one or more constraints on what should and should not be in the final fact-list.

Background knowledge in A3 includes information similar to *deffacts* and the initial facts associated with an instance. CR2 is able to define certain background information as being appropriate only for specific instances. In the original implementation of A3, all background facts must be considered during the proof of each example.

Both A3 and CR2 are able to use variable typing constraints that restrict allowable revisions by defining, for example, that a literal or CE that compares *row* and *file* (for a chess domain) are inappropriate. A3's version of variable typing is more expressive than CR2's version. CR2's main use of variable typing is to disallow the use of potentially non-numeric variables in test CEs.

Both systems also rely heavily on accuracy/error over the examples and constraints as the main evaluation metric when selecting between revisions.

In terms of structure, A3 is also similar to CR2 in that it has an implicit SPR-like component, FIND_BEST_ASSUMPTIONS and MODIFY_THEORY, that identifies the best minimal revision to the current theory. The procedure, FIND_BEST_ASSUMPTIONS, identifies assumptions about what should and should not be provable by the theory that if true would allow incorrectly classified examples to be correctly classified. MODIFY_THEORY uses the best assumption found by FIND_BEST_ASSUMPTIONS to determine points in the theory that should be revised and what types of revision to perform. MODIFY_THEORY uses the best evaluated revision to revise the theory.

The ASSUME RIO technique in CR2 is probably most analogous to the procedure FIND_BEST_ASSUMPTIONS (and a portion of MODIFY_THEORY). Like FIND_BEST_ASSUMPTIONS, ASSUME bases it's identification of potential revisions on violated constraints and both techniques use "What if?" assumptions to identify possible revision points and revisions to the rulebase.

Unlike FIND_BEST_ASSUMPTIONS and MODIFY_THEORY, the RIO techniques (including ASSUME) identify an ordered set of high-level revisions which are expanded and evaluated in order using SPR's revision operators. The procedure FIND_BEST_ASSUMPTIONS identifies a single best assumption which, via MODIFY_THEORY, identifies an unordered set of high-level revisions which are expanded and evaluated using the revision operators. Both algorithms use, respectively, examples and CVs to select and order the revisions.

Given the best assumption produced by FIND_BEST_ASSUMPTIONS, the procedure MODIFY_THEORY identifies all clauses that used the assumption and all clauses in the concept associated with the assumption. Clause specialization or generalization is performed on clauses that used the assumption. Clause specialization is performed when the literal in the clause associated with the assumption has the same sign as the assumption. Clause generalization is performs on all other clauses that used the assumption. Concept specialization or generalization is performed on the assumption's concept's clauses. Concept specialization is performed if the assumption is negated, otherwise concept generalization is performed.

What A3 calls clause specialization is fairly different from what CR2 calls rule specialization. Clause specialization is done in the context of an assumption that was used by a literal in the clause. The three forms of clause specialization used by A3 include: negation of the assumption's literal (not strict specialization), addition of one or more new literals to the clause, and deletion of the clause from the theory. With respect to the last two forms, CR2 is able to both add a single

CE to a rule and delete an entire rule. In addition, if one looks at a production rule with multiple asserts as multiple Horn clauses, CR2 is also able to delete a clause by deleting one of the assert actions.

What A3 calls clause generalization is also fairly different from what CR2 calls rule generalization. As with clause specialization, clause generalization is done in the context of an assumption that was used by a literal in the clause. The three forms of clause generalization used by A3 include: deletion of the literal associated with the assumptions, replacement of the literal with an induced set of new literals (not strict generalization) and induction of a new clause. CR2 only performs CE deletion when generalizing a rule.

Concept specialization and generalization in A3 does many of the same revisions that were described above. Concept specialization identifies the best clause in the concept associated with the assumption and either deletes it or specializes it. Concept generalization attempts to learn a new clause for the concept.

Clause induction in A3 is done using a FOIL-like hill-climbing search that adds literals as long as evaluation improves. The only slightly analogous technique that CR2 has to perform rule induction is the addition of new assert actions to existing rules.

8.4.2 Empirical Comparison

This section will show an empirical comparison of CR2 and A3 using the student loan and king-rook-king domains described previously in this chapter.

Student Loan

The first comparison between CR2 and A3 will be done using the four mutation version of the student loan rulebase. This mutated rulebase has a missing clause which CR2 should have difficulty revising because it is not capable of rule induction. Figure 8.1 shows average CV error as a function of training set size for CR2 and A3. CR2 was run using an rlimit of 25 and without an rlimit. These curves show that CR2 performs similarly even though it cannot directly handle the missing clause problem. One point to be noted is that neither algorithm produces 100% accurate results. Therefore, even though A3 is capable of clause induction, its blame assignment mechanism may not be identifying the correct revisions.

In addition, an analysis of individual runs of CR2 on this rulebase showed that CR2 has an indirect solution to inducing the missing rule. As one of the

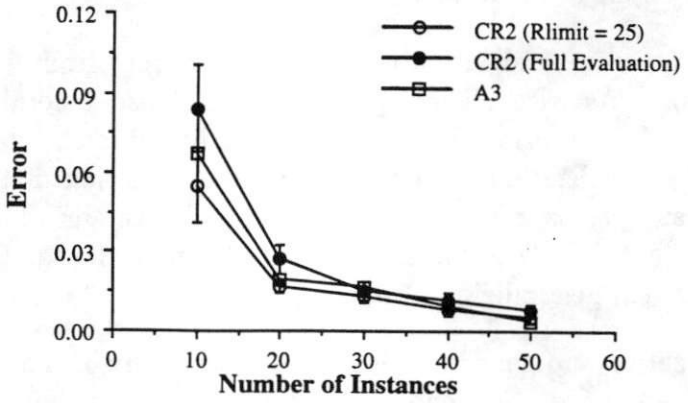


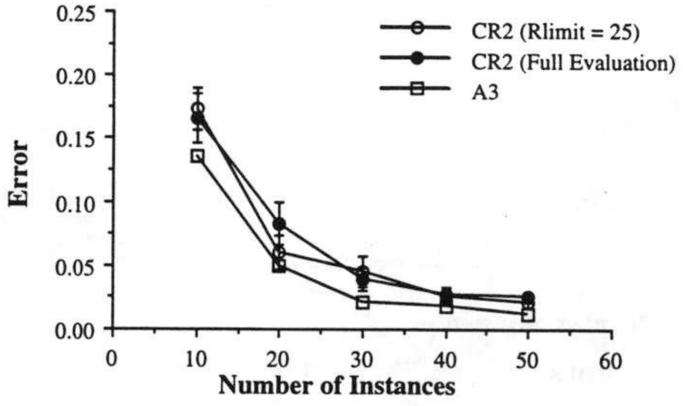
Figure 8.1: Average CV error of revised four mutation rulebase as a function of the number of training instances for the student loan domain when revised using CR2 and A3. Eight intermediate concepts.

mutations to this rulebase is an extra rule and the fact being asserted by the extra rule is the same as the fact that should be asserted by the missing rule, CR2 takes the approach of specializing and then generalizing the extra rule until it looks similar to the missing rule. It first specializes the extra rule with a CE present in the missing rule. Specialization reduces the errors caused by the extra rule's inappropriate firing. Next it generalizes the rule by removing CEs that are not present in the missing rule. Within a couple of revisions, the extra rule looks identical to the missing rule. Of course, if the extra rule had not had the same assert as the missing rule, CR2 would probably not have been able to perform this work around.

The curves shown in Figure 8.1 are based on the use of the target concept (`no-payment-due`) and all of the remaining (8) intermediate concept as constraints and examples. As the number of intermediate concepts is reduced, both algorithms perform more poorly. Figure 8.2 shows two graphs, each constructed similarly to the graph in Figure 8.1, except that these graphs were produced using less of the intermediate concepts. At smaller numbers of the intermediate concepts, A3 seems to do better than CR2. It is likely that CR2 has less direction toward solving the missing clause problem when there are fewer intermediate concept constraints.

The next set of comparisons using the student loan domain is based on the use of the three mutation rulebase. Figure 8.3 shows that CR2 still does more

A. No Intermediate Concepts



B. 2 Intermediate Concepts

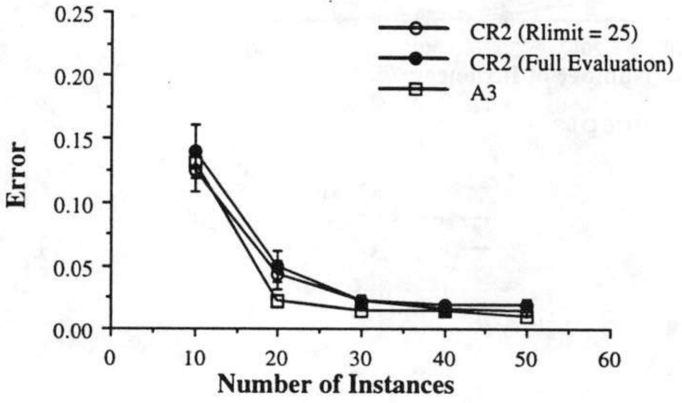
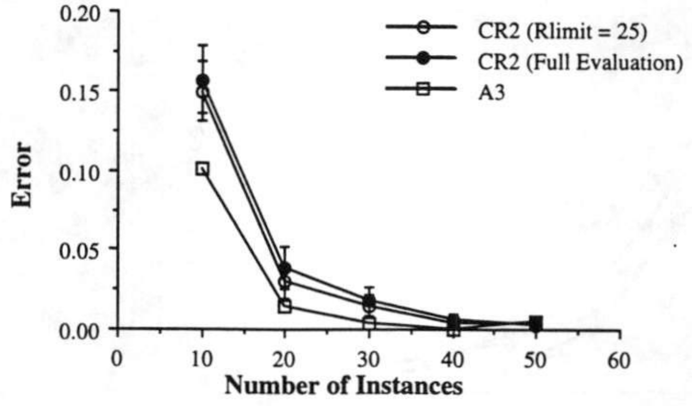
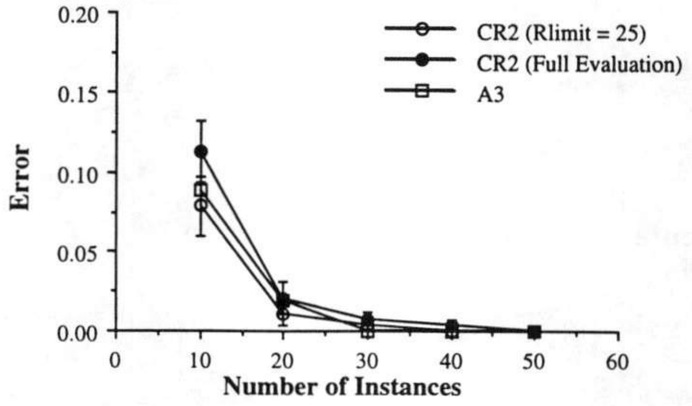


Figure 8.2: Average CV error of revised four mutation rulebase as a function of the number of training instances for the student loan domain when revised using CR2 and A3. No intermediate concepts and two intermediate concepts.

A. No Intermediate Concepts



B. 2 Intermediate Concepts



C. 8 Intermediate Concepts

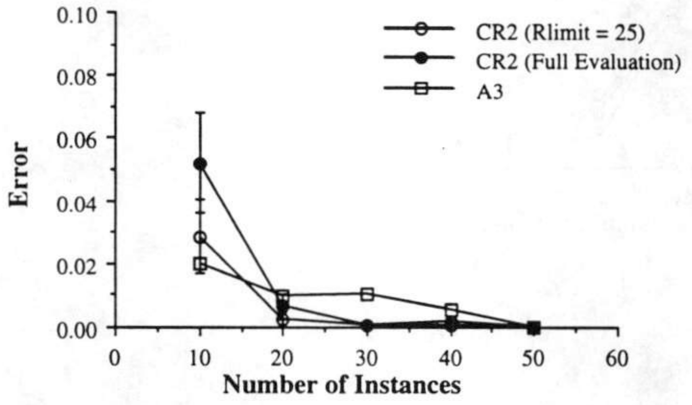


Figure 8.3: Average CV error of revised three mutation rulebase as a function of the number of training instances for the student loan domain when revised using CR2 and A3. No intermediate concepts and two intermediate concepts.

poorly when there are no intermediate concepts but does as well if not better than A3 (for full evaluation) at two and eight intermediate concepts.

As a final note before going on, it is interesting that at only 25 low-level revision evaluations, CR2 does nearly as well as A3. When the limit was lifted, CR2 sometimes appeared to perform better. In terms of cpu-time, A3 runs about an order of magnitude slower than CR2 on this and other domains that were used for comparison.

King-Rook-King

The comparison of CR2 and A3 on the king-rook-king domain highlights many differences between these two systems. These differences include:

- variable typing capability and expressiveness
- limiting the number of new variables introduced during rule specialization
- limiting the number of revision evaluations (rlimit)

The first of these differences, variable typing, was described previously and is an important difference for this domain. While CR2 can be told that an argument in a CE should be bound to only numbers, A3 is capable of being told that such an argument should only be bound to certain types of number, e.g. the *rank* or *file* of a piece on a chess board. CR2's inability to use more detailed forms of typing allows it to consider and possibly accept revisions that were semantic non-sense (at least for this domain), e.g the rank of the white king is less than the file of the black king.

The second of these differences is also a capability of A3. However, it is more an efficiency hack than a principled design decision. Limiting the number of new variables available for introduction during clause specialization was originally used by FOIL (Quinlan, 1990) as a method for improving accuracy and speed during the rule induction. Limiting the number of new variables reduces the search space and if one knows the appropriate limit for a particular problem, refines the hypothesis space and improves accuracy. Unfortunately, the appropriate limit is never known except for artificial problems. For A3 and the king-rook-king domain, limiting new variables gave A3 an advantage over CR2 by significantly reducing the possibility of specializing a rule using the arity six, `position` literal. Since this literal was not missing from any clause in the mutated rulebase, A3's inability to use it could only have benefited its performance. A3's default limit of only allowing the introduction of three new variables per hill-climbing step was used during runs on this domain.

The third and last major difference is a capability of CR2 only that allows it to evaluate only a subset of all identified high-level revisions (rlimit). This capability, while not necessarily allowing CR2 to perform more accurately than A3, does provide some insights into how expensive it would have been to allow A3 to be able to introduce an unlimited number of new variables during specialization of the king-rook-king domain. In addition, it shows that much of the productive evaluation of revisions, by CR2, is done quite early.

As for the comparison of these two algorithms on the king-rook-king domain, Figure 8.4 shows error as a function of training set size for CR2 and A3 (curves for CR2 are shown for an rlimit of 25 and 100,000). From the graphs, A3 appears to do much better than CR2. The combination of enhanced variable typing and the inability to specialize with `position/6` likely contributes to this result. Note the similarity in performance by CR2 when the rlimit is 25 and when the rlimit is 100,000. Most of the effective work that was done was done early.

8.5 Other Related Systems

This section presents four related theory revision systems that share some similarities with CR2. Each system is briefly described and compared to CR2 and to one another.

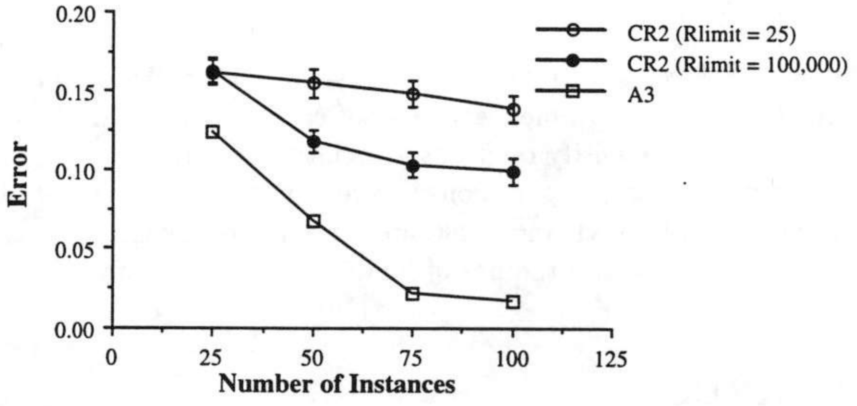
8.5.1 CLIPS-R

CLIPS-R is a forerunner to CR2. It was an analysis of empirical results using CLIPS-R that led to the recognition that research into a more refined system was needed.

CLIPS-R is a production system rulebase reviser that is able to revise rulebases similar to the rulebases revised by CR2. CLIPS-R is actually able to revise a slightly broader class of rulebases. For example, CLIPS-R can revise rulebases that include `if-than-else` actions. CLIPS-R is also able to use a different class of constraints than CR2. CLIPS-R is able to use constraints on the ordering of observable actions (e.g. printout actions). In addition, CLIPS-R includes a rule induction component.

Unfortunately, CLIPS-R performs a very undirected search to find the best revision. It uses a generalization of the ASSUME technique to identify likely revisions. This technique has the ability to identify rules that did not fire but could

A. No Intermediate Concepts



B. 3 Intermediate Concepts

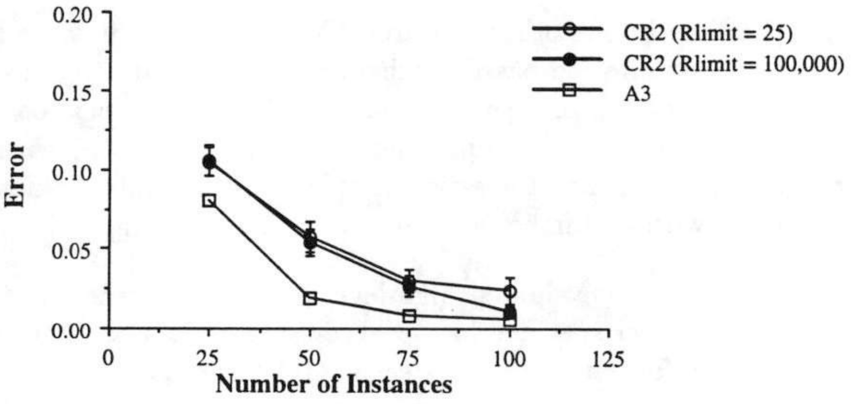


Figure 8.4: Average CV error of revised rulebase as a function of the number of training instances for the king-rook-king domain when revised using CR2 and A3. No intermediate concepts and three intermediate concepts.

have asserted/retracted a missing/extra fact. Empirical results showed that this approach ended up generating revisions to decrease more than half of the rules in a rulebase. Since CLIPS-R included no approach for ordering revisions, it often evaluated more than half of the potential single revision search space in order to identify the best evaluated revision.

Since CLIPS-R does not include the rich set of RIO techniques that CR2 includes, much of the revision problem space is not covered using CLIPS-R. CLIPS-R is only guaranteed to identify revisions associated with those partitions covered by ASSUME and portions of the partitions covered by INCREASE. Except by chance, the partitions handled by EXP and MISC are not handled by CLIPS-R. Note, the above discussion assumes only the use of final fact-list constraints.

8.5.2 CLARUS

CLARUS is a theory revision system that is very much like A3. The main difference between CLARUS and A3 is that CLARUS is able to take advantage of linguistic-based semantics concerning the meaning and relationships between terms in the theory and background knowledge. These relationships are used by CLARUS to guide the selection of a best revision after improvement in accuracy has been considered.

CLARUS differs in one other way from A3 that makes it more like CR2. CLARUS uses the assumption-based mechanism of A3 to identify, not the best assumption, but an ordering of assumptions. The revisions associated with each assumption are evaluated in order until the evaluation associated with the best evaluated revision is better than the estimate of the best possible evaluation of the revisions associated with all further assumptions in the ordering.

CR2 also orders and evaluated high-level revisions in order, but does not halt evaluation until a user specified low-level revision evaluation limit is reached. The techniques and information used by each RIO technique to order high-level revisions is too diverse and does not lend itself to a heuristic that would allow it to know that further expansion and evaluation of high-level revisions can provide no more decrease in error.

8.5.3 FORTE

FORTE, like A3 and CLARUS, is able to revise relational Horn clause theories. However, unlike these other two algorithms, FORTE is not able to revise

Table 8.3. SEEK2 Rulebase Characteristics.

- propositional rules that have confidence factors associated with their conclusions,
- have no retract actions or salience values,
- may have a choice-number which tells the minimum number of conditions on the left side of a rule that must be satisfied to assert the conclusion.
- an extra rule

Horn clause theories that include negated literals. FORTE takes as input a set of positive and negative examples for concepts in the theory like A3, and has the same general hill-climbing organization as A3.

Very similar to CR2, FORTE uses misclassified examples to directly identify revision points in the theory. Revision points correspond to calls to either specialize or generalize the theory at these points. Recall that A3 and CLARUS use misclassified examples to identify assumptions which indirectly identify points in the theory in need of revision. Specialization revision points are associated with clauses that participated in the proof of a negative example. Generalization revision points correspond to specific literals in clauses, clauses and entire predicates and are identified by the failure to prove positive examples.

FORTE is best described as an operator-based revision system because it has such a wide variety of operators. Depending on the type and location of a revision point, some of the potential operators that may be called include: delete rule, delete antecedent, add antecedent, add rule, identification and absorption. The later two operators are inverse resolution operators that were presented in CIGOL (Muggleton & Buntine, 1988). Both of these operators have the potential of producing new clauses (generalizing the theory).

8.5.4 SEEK2

SEEK2 is an older theory revision system that is able to revise production system rulebases. The type of production system rulebases that SEEK2 is able to revise, however, differs significantly from the kind of rulebases that CR2 is able to revise. The production system rulebases revisable by SEEK2 have the form shown in Table 8.3. In many ways, the rules revised by SEEK2 are simply Horn clauses that are executed in a forward chaining manner.

Like CR2, SEEK2 uses a set of cases that describe what conclusions should be reached by the rulebase given a certain set of background facts. When running in "automatic pilot mode", SEEK2 is a hill-climbing systems. SEEK2 first identifies an ordering of conclusions to revise. That is, SEEK2 evaluates each case and rates each expected conclusion by the number of "false positives" and "false negatives" that were made. This rating is used to identify which conclusions are most in need of improvement. Once a conclusion is selected, heuristics that are based on statistics associated with the execution of the cases are used to identify potential revisions. The potential revision operators include the addition and deletion of conditions and the modification of the choice-number.

8.6 Chapter Summary

This chapter presented a comparison between CR2 and four other theory revision systems. A detailed analytical and empirical comparison was done between CR2 and A3.

Analytical comparison between CR2 and A3, CLARUS and FORTE showed that CR2 is similar, in terms of high-level structure, to existing Horn clause based revision systems. At a lower level, the design of CR2 is much more involved because of the added complexities of dealing with production system rulebases. The main source of information used to guide the revision process, final fact-list constraints, is similar to the examples used by Horn clause based systems.

Comparisons to CLIPS-R showed that CLIPS-R is far less refined than CR2, and does not cover as much of the revision problem space as CR2 when only final fact-list constraints are used. Relative to CR2, CLIPS-R is computationally much more expensive than CR2.

An empirical comparison was done between CR2 and A3 using two mutated versions of the student loan domain and one mutated version of the king-rook-king domain. These comparisons showed that, for rulebases capable of being represented as Horn clause theories, in many cases, CR2 did as well if not better than A3, but in other cases, much worse than A3. Where differences in performance occurred, explanations were presented in terms of differences in the design of the two algorithms. For example, as CR2 does not include a rule induction algorithm (that A3 does include), when a small number of intermediate concepts were available, CR2 performs more poorly than A3 on the mutated version of the student loan domain that includes a missing rule mutation. However, at higher numbers of intermediate concepts, CR2 was able to deal with the missing rule mutation by converting an extra rule with the same consequence as the missing rule into the missing rule.

Chapter 9
Conclusion

9.1 Chapter Overview

This chapter presents an overview of this research. Included here is a summary of its main contributions, as well as a description of some of its limitations and suggestions for future research.

9.2 Contributions

A major trend in theory revision research has been the revision of more expressive and different forms of rulebases. This research has followed that trend by exploring issues that involve the revision of production system rulebases. Most theory revision research has focused on the revision of Horn clause rulebases.

The main contributions of this research revolve around the revision of production system rulebases. One of the reasons the revision of production system rulebases is an important goal is because production system rulebases tend to be the representation of choice for expert system knowledge bases. With respect to Horn clause theories, production system rulebases are suitable for a wider variety of tasks. For example, while Horn clause based knowledge bases tend to be useful only for logical classification tasks, production system rulebases are able to more easily perform procedural tasks including planning, monitoring and control. Production system rulebases also tend to be more efficient, in that information needed for multiple tasks may be computed once and asserted to the fact-list. The ability to finely control when production system rules fire by asserting and retracting enabling facts is another reason that production system rulebases are efficient. The non-monotonic capabilities of production system rulebases also allows them to perform reasoning that Horn clause based systems are not capable of.

Research into the revision of production system rulebases is non-trivial. One reason for this includes the wide variety of characteristics and capabilities of production rules, e.g. multiple actions per rule, assert and retract actions and rule salience. Another reason lies in the difficulty in identifying likely revisions. As this research focused on the use of information that is similar to the information that is commonly used to guide Horn clause based revision systems, e.g final fact-list constraints, the non-monotonic nature of the contents of the fact-list makes the identification of likely revisions an extremely difficult task. In addition, since there tends to be a wide variety of ways to encode knowledge in production system rulebases, knowledge engineers who design production system knowledge bases tend to follow organizational styles that can be exploited during the revision process.

The individual contributions of this research includes a revision problem space model that is used to characterize revision problems, revision identification and ordering (RIO) techniques that are designed to cover most areas of the revision problem space, a new set of operators for the revision of production system rules, an approach that identifies and avoids revisions that would produce ill-structured rules, and a technique for placing a resource limit of the number of revision evaluations that are made during each hill-climbing step. Each of these contributions is discussed below.

9.2.1 Revision Problem Space Model

The revision problem space was formally defined because of difficulties in designing a single general purpose algorithm that could identify the correct revision independent of what the problem was. As it became clear that no single approach would solve the problem, a characterization of the revision task faced by the reviser was formalized. This model became what is called the revision problem space model (see Chapter 3 for more details).

The revision problem space model is composed of a set of features and constraints on the features. The model assumes that a single problem exists in the rulebase and final fact-list constraints are available to guide the identification of the correct revision. The revision problem space model allows the space to be partitioned, as was shown in Figure 4.2, so that a better understanding and description of the types of problems faced by a revision system are possible.

The formal definition of a revision problem space model for theory revision research is important because it allows for a case by case analysis of how well a theory revision system does.

Most Horn clause based theory revision research has not addressed the task of producing a formal revision problem space model. One reason for this is the relative simplicity of the Horn clause rulebase revision task. Some systems, however, have identified simple implicitly defined models that are encoded in their blame assignment techniques. The only features that are used to describe these models are the appropriate or inappropriate provability of an example, the problem type and location. By not formally defining a revision problem space model, many revision systems have fallen short of being able to accurately characterize when their systems will not work.

In this research, the revision problem space model was used to understand where new blame assignment (RIO) techniques were needed. The model also showed which partitions of the revision problem space were not covered by any technique. The utility of the model was made clear during the empirical analyses in Chapter 5. There, for artificially mutated rulebases, the mapping between the revision problems associated with each mutated rulebase and the RIO techniques were used to produce RIO distributions. These RIO distributions showed which mutated rulebases needed which RIO techniques. Ablation studies that omitted the use of certain RIO techniques and the unrevisability of some mutated rulebases validated the utility of the revision problem space model.

9.2.2 RIO Techniques

The RIO (revision identification and ordering) techniques were defined as an independent set of procedures because of the difficulties in dealing with the revision problem space as a whole. Each RIO technique was designed to cover some uncovered partition of the revision problem space. The availability of the revision problem space model was key to understanding what partitions remained in need of coverage.

Each RIO technique is independent of every other RIO technique. Each operates under the assumption that the revision problem that it is to deal with is appropriate for it. This is not always the case. When it is the case, they are designed to identify the correct revision and place it near the top of an ordering of identified revisions. Each technique generates its own ordering of identified revisions. When the RIO technique is inappropriate for a particular revision problem, a random set of revisions is identified and placed in the ordering. After all RIO techniques have been executed, a merged ordering of revisions is produced from the original orderings. Revisions that were near the tops of the individual orderings are placed near the top of the merged ordering. The revisions in the merged ordering are evaluated by a set of revision operators in order.

The approach of using an independent set of RIO techniques and merging the results is new to theory revision research and to some may be looked at as a step backwards. Most researchers have taken the approach of designing a single technique that uses the information it has from the examples to identify the correct revision. Such techniques rarely cover the entire revision problem space. Intuitively a more integrate approach would have been better for this research. Unfortunately, the revision problem space for production system rulebases is just too complex to be handled by a single technique. Four different techniques were designed for this research and are briefly reviewed below.

The ASSUME RIO technique is probably the most directed and accurate technique. It is similar to existing Horn clause based blame assignment techniques because it uses traces of rule firings in the same way that Horn clause based systems use proofs and partial proofs to look for potential revisions. Unlike Horn clause based systems, however, ASSUME had to be extended in order to deal with the issues of rule salience, multiple actions per rule and retract actions. The ASSUME RIO technique identifies decrease rule, and add and delete action revisions.

The EXP RIO technique was designed as an aid to the ASSUME RIO technique. It can identify a subset of the revisions that ASSUME identifies, but can do so under conditions that ASSUME cannot. For example, EXP is designed to identify decrease rule revisions when there is no connection, via trace information, between the CV and the problem location. ASSUME requires a traversal of trace information to identify decrease rule revisions. EXP identifies revisions by running simple experiments that re-execute errorful instances. EXP identifies decrease rule and delete action revisions.

The INCREASE RIO technique identifies an increase rule revision for each rule in the rulebase. It is the only RIO technique that can identify increase rule revisions. INCREASE is a brute force technique in that it identifies the entire space of such revisions. It uses trace information to order the identified revisions. The design of INCREASE as a brute force approach was needed to ensure complete coverage of the space. Most Horn clause based theory revision systems use a more directed approach to identify generalization revisions. Unfortunately, these approaches are often not capable of identifying the correct clause to revise because of a lack of connection between the proof tree and the problem clause. Since the number of revisions identified by INCREASE is linear to the number of rules in the rulebase, the brute force nature of the technique is not a significant computational problem.

The last RIO technique covers an area of the revision problem space where there is little information to guide the identification of the correct revision. Like INCREASE, the MISC RIO technique is also a brute force technique. It identifies a delete action revision for each action in each rule and an add retract revision for each positive pattern CE in each rule. As there are few actions and CEs in each rule, this is also a computationally efficient technique. Overall rule error over the instances is used to order these revisions.

One partition of the revision problem space is covered by no RIO technique. This partition corresponds to the leaf labeled uncovered in Figure 4.2. This partition is similar to the partition that MISC was designed to cover except that the problem type is missing assert. No computationally efficient RIO technique could be designed to cover this partition. Consequently, if a revision system is to handle this partition, constraints, other than final fact-list constraints, will be needed.

9.2.3 Revision Operators

An enhanced set of revision operators had to be designed in order to deal with the added complexities of production rules. New operators were required to allow revisions to rule salience and addition and deletion of actions. Modifications to rule generalization were also required in order to deal with dependencies in other parts of the rule. Most Horn clause based revision systems have only rule specialization, generalization and induction operators. Some existing techniques do use other types of operators, e.g. inverting resolution, but they are rare.

The operators that increase or decrease rule salience were designed to revise salience such that only a minimal number of operator instances needed to be evaluated. Depending on whether an increase of decrease in salience was required, representative new saliences were identified by looking at the set of different saliences used in the rulebase.

The specialization and generalization operators added and deleted, respectively, single CEs. Before deleting a CE, the operator had to consider whether variables introduced by the CE were required by test CEs or assert actions. Variable used in both test CEs and assert actions must be bound prior to use.

9.2.4 Rule Structure Filtering

Rule structure filtering is another important contribution of this research. It represents a move away from revision systems that only revise based on accuracy over the instances. Rule structure filtering implements a bias for revisions that do not produce ill-structured rules.

Rule structure filtering takes advantage of the fact that each rulebase that is produced by a knowledge engineer tends to be organized so that it is easy to understand. Rule structure filtering takes into consideration this organization in the form of rule structure. For example, if the knowledge engineer chooses to implement the knowledge base without the use of different saliences, rule structure filtering would likely see revisions that change salience as producing ill-structured rules. Rule structure filtering goes beyond considering what global characteristics exist in the rulebase, however. It sees the rulebase as a set of rule groups, where rules are grouped by structural similarity.

Rule structure filtering is applied to the set of best evaluated revisions prior to selection of the best revision to make to the rulebase. Rule structure filtering looks at a rule being revised and identifies the best group of rules consistent with the rule. It then revises the rule and again identifies the best group consistent with the rule. If the "quality" of the revised rule's best group is worse than the quality of the unrevised rule's best group, the revision is removed. Quality is a function of the size of the group and the size of the group's description. Larger groups with many characteristics in common are preferred.

Rule structure filtering was empirically shown in Chapter 7 to be an appropriate bias for at least 80% of the mutated rulebases used. Error rate comparisons between using and not using rule structure filtering showed that it improved accuracy for low training set sizes (when more revisions evaluate similarly). At higher training set sizes, rule structure filtering did not hurt performance.

9.2.5 Limiting Resources

One problem with many areas of artificial intelligence research is the incredible cost in computational resources. Theory revision research is no exception. One important contribution of this research is the study of an approach for limiting the number of revision evaluations that are made during each hill-climbing step.

This approach takes advantage of the ordering of revisions produced by the RIO techniques. Revisions are evaluated one at a time and in order until a user defined limit on the number of revision evaluations is reached. The best evaluated revisions are passed to rule structure filtering. This approach makes intractable revision problems tractable.

The main results of this study showed that for many rulebases, limiting resources to less than 100 evaluations often produced results that were not much less accurate than would be achieved for full evaluation (more than 1000 evaluations).

In general, limiting evaluations had the effect of increasing asymptotic error (as training set size increased).

9.3 Limitations & Future Work

As a research system, it has been demonstrated repeatedly that CR2 has succeeded at its goal of being able to revise production system rulebases. This research has also demonstrated some of the difficulties in trying to revise these rulebases using only final fact-list constraints. As the revision of production system rulebases is a new area, there are still a number of unexplored issues.

Some of the limitation and areas for future work are described below. They include enhancement of the revision problem space model, construction of a rule induction component, the use of "rulebase understanding" techniques, and the use of new forms of constraints and biases.

9.3.1 Enhanced Revision Problem Space Model

The revision problem space model is based on a well defined set of assumptions. One way to enhance this model is to generalize these assumptions. For example, one of the assumptions constrains the types of problems that a rulebase can have. The current revision problem space model is designed to revise rulebases that have either a missing or extra CE, a missing or extra assert or retract action, a wrong salience or an extra rule. Notably missing from this set of problem types is *missing rule* (to be discussed later). Another assumption constrains the rulebase to have only a single problem. Relaxation of this assumption could mean removal of the limit entirely or removal of the limit under specific circumstances. For example, certain combinations of problems might be allowed.

With respect to the possibility of removing the limit completely, it is simply not practical. The model would be so complex that it would be useless. As it is now, revision, using only final fact-list constraints, is so difficult that certain partitions of the space are only coverable by weak (brute force) techniques, or no technique at all.

However, loosening the constraint so that certain combination of problem classes are allowed does appear to have potential. For example, as it is currently, the revision problem space model could allow multiple missing CEs in a rule or multiple extra CEs in a rule without much change. However, the RIO techniques

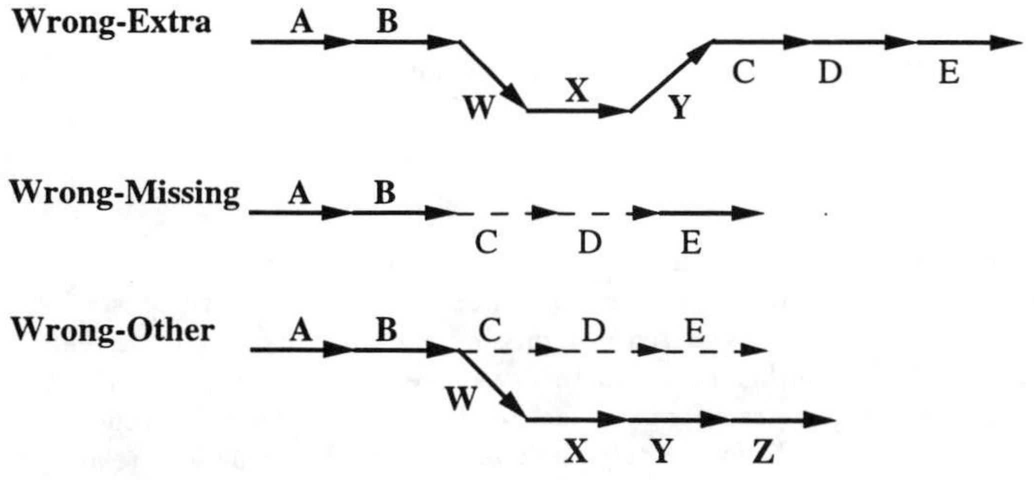


Figure 9.1. Expansion of wrong rule firing sequence deviation class.

and operators would have to change. In addition, multiple extra actions in a rule associated with areas of the revision problem space handled by MISC could also be handled using the current model.

Another way that the revision problem space model could be enhanced is through an extension of the rule firing sequence deviation classes. A large percentage of the singly mutated rulebases that were generated for this research executed the *wrong* rule firing sequence deviation class. It might be worth while to refine the *wrong* class into multiple, more specific, classes so that better RIO techniques could be associated with the more specific classes. For example, the "wrong" rule firing sequence deviation class could be divided into the three cases shown in Figure 9.1.

The first new class, *wrong-extra*, is the sequence of activation firings that execute correctly up to a point, execute one or more extra activations and then execute the remainder of the correct sequence. The second new class, *wrong-missing*, is the sequence of activation firings that execute correctly up to a point, omit the execution of one or more activations, and complete the remainder of the correct sequence. The last class is the class of all other wrong activation sequences.

9.3.2 Rule Induction

Notably missing from this research is a rule induction mechanism. The main reason that no such mechanism was designed is because it is not a problem that can be adequately solved using only final fact-list constraints.

For example, to induce a rule, a revision system has to know what actions the rule should execute and where the rule should fire. Using final fact-list constraints, one scenario is that a CV was caused by a missing rule that should have fired after execution of all of the other rules. In this case, we know the action to execute (assert for missing fact, retract for extra fact) and we know under which conditions the new rule should fire. Unfortunately, the possibility of this scenario is relatively small. A more likely scenario is that the CV was directly caused because the missing rule should have been executed instead of some other rule from some non-final fact-list. For this case, we only know the action to execute. An even more likely situation is that the CV was caused by a missing rule that should have fired from some non-final fact-list and should have enabled the execution of a chain of activation that ended up satisfying the violated constraint. Of course, in this last case, we don't even know what action(s) the rule should execute.

Compared to the revision of production system rulebases, rule induction for Horn clause based revision systems is much easier. One reason for this is that Horn clauses have only a single action (analogous to an assert). Another reason has to do with the monotonic nature of Horn clause rulebases. Most revisions to a Horn clause rulebase can be identified using a single ASSUME-like technique.

In general, rule induction cannot be adequately performed using only final fact-list constraints. Any such attempt would be only a token gesture. If, on the other hand, one knew what actions the rule should execute and where (from which fact-list) the rule should execute, a potential rule induction algorithm could create a rule with CEs based on the facts in the fact-list. A localized hill climbing search that had operators that could remove and generalize existing positive pattern CEs and add negative pattern CEs and positive and negative test CEs could be used to refine the new rule. If it was known where a new rule should fire across multiple instances, the LGG of the facts in the fact-lists associated with the instances could be used to form a better initial set of pattern CEs for the new rule. The later technique was used in CLIPS-R.

9.3.3 Rulebase Understanding

Understanding a rulebase is obviously an important step in order to revise it. Humans are able to take advantage of both semantic and structural information when trying to understand rulebases. An area of future work for the revision of production system rulebases is to use structural and execution characteristics of the rulebase (and instances) to form an understanding.

Based on the experience of reviewing and manually revising multiple rulebases, production system rulebases have many structural characteristics that can be taken advantage of when trying to understand them. For example, the approach of retracting facts after use (as is done in the nematode domain) is one approach to cleaning up the fact-list so that the group of rules that asserted and used the facts may be executed again. Another approach that may signal the same situation is a "clean-up rule" that fires after execution of a group of rules. The clean-up rule would have high salience and would have retracts for facts asserted by the group.

Another such example is the "default rule" used in the auto diagnosis rulebase. This rule is structurally unique. It has the same negated pattern CE as many other rules in the rulebase, but has a low salience. The purpose of this rule is to provide a default diagnosis under the condition that no other rule has asserted a diagnostic fact to the fact-list. Having low salience, it remains at the bottom of the agenda until a diagnosis by another rule deactivates it (by matching the negated pattern CE) or until the execution of all other diagnostic rules are exhausted. The presence of such a rule signals a classification task.

In addition to using rule structure to understand a rulebase, an understanding of the purpose of facts, asserted to the fact-list, can prove useful. For example, the purpose of many groups of rules is to produce a product. In the previous example, a look at the assert action in the "default rule" provides an example of the product of the classification task, e.g. `(repair ..)`. Also, many rules are enabled by the presence of "control facts" that are asserted and retracted when the firing of certain groups of rules is desired. Another class of facts are facts that are used purely for inferencing.

Beyond structural characteristics, execution of the rulebase using the instances is another technique for understanding it. Using the traces of error-free instances, the correct traversal of rule groups and generation of data products can be determined and used to better understand the rulebase.

In general, the main difficulty for rulebase understanding is the challenge of merging each of these sources of information into a useful model. Any solution to this task is likely to be conceptually messy. It may be possible to use a maximum likelihood-like technique for this purpose.

9.3.4 New Forms of Constraints and Biases

Another potential area of future research is the use of new forms of constraints and biases to guide the revision process. CR2 was designed to use final fact-list constraints and constraints on the number of activation executions per instance.

Final fact-list constraints represent a modest step away from the constraints used by most Horn clause based revision systems. Unfortunately, the use of only final fact-list constraints led to an uncovered portion of the revision problem space. Since evidence based on the revision of the nematode identification rulebase (Chapter 5) showed that many revision problems correspond to this uncovered partition, future research in the area of production system revision will need to use additional forms of constraints.

Some potential forms of constraints and biases that deserve further research include a user provided partitioning of the rulebase into rule groups and constraints on contents of the fact-list upon exit from rule groups, constraints on the order of execution of certain observable actions (e.g. printout actions) and any information that would help rulebase understanding techniques.

If a user could provide an initial partitioning of the rulebase into rule groups and constraints on the fact-list upon exit from these groups, the revision task can be broken down into the revision of individual groups of rules. Since each group is likely to have a simpler purpose and less variety of structure, revision should be easier because more of the CVs are likely to be traceable back to the problem rule.

Constraints on the ordering of observable action executions is a constraint that is very different from fact-list constraints. Such constraints can tell the revision system that some printout action should only be executed once per instance or that the printing of the "Welcome!" message should proceed the printing of the diagnosis. Ordering constraints on observable actions would be very important during the revision of rulebases that implement monitoring and control systems.

In general, this research showed that final fact-list constraints did not provide enough guidance to adequately identify likely revision when the CV location was in a rule that did not fire because of an earlier problem in the execution. Each of the above mentioned forms of constraints has the potential to narrow the search for problem rules by pointing out that a problem exists earlier in the activation firing sequence than the final fact-list.

9.3.5 Non-Deterministic Rule Execution Model

The production system execution model described in Chapter 2 is non-deterministic in terms of the order of rule execution. This occurs because the model does not define the order in which activations from the same activation execution and with the same salience are placed on the agenda. This is important because the order that activations are placed on the agenda defines the order in which activations are executed.

For CR2, this problem was satisfactorily solved by ordering the newly produced activations prior to adding them to the agenda. The ordering method used was based on the order in which the rule associated with the activation was loaded into the internal rulebase. This ordering is referred to as *source file ordering.* Activations associated with rules that are loaded first are placed highest in the ordering. Note, the production system execution model used by CR2 is still incomplete because the ordering of multiple new activations that are associated with the same rule is still undefined. Any complete model would have to also use the bindings associated with an activation to define the ordering.

There are two reasons why the source file ordering method used by CR2 is satisfactory. First, and of least importance, for most of the rulebases used in this research, multiple activations with the same rule are never produced at the same time. Only the nematode identification domain generates multiple activations for the same rule. Luckily, it does so at a time when the ordering of rule executions does not matter.

The second reason is more important and has to do with the problem types (mutations) used in this research. First, none of the problem types correspond to a direct change in the source file ordering. Second, the *missing rule* problem type has not been studied in this research. If rule induction had been required to repair certain mutated rulebases, then, in addition to identifying the rule to add to the rulebase, the location within the source file to place the new rule would also have to be determined.

To completely solve the production system revision problem, the ordering of rule executions must be completely defined and modifiable by the revision system. Fully ordering activations prior to adding them to the agenda is one solution. Future research is needed to identify a method for ordering activations based on their bindings. If such an ordering method used the ordering of values bound to variables introduced by positive pattern CEs in the rule, in addition to requiring operators that could revise the order of rules in the source file, a revision system would also require operators that could change of the ordering of positive pattern CEs in a rule.

9.3.6 CR2 as an Industrial Tool

As an industrial tool, the design of CR2 is still too fragile. Among other reasons, final fact-list constraints are not adequate for a rule induction component. New forms of constraints that can be easily attained in practice are needed.

Accordingly, an interesting area of future work which has the potential of making CR2 a practical tool is the construction of an interactive front-end. The front-end could be used to gather constraints, in the form of interactive user feedback, on how instances are doing as they execute. For example, as an instance executes, the user might signal that some fact is missing from the current fact-list or that something was just printed that shouldn't have been.

If such a front-end was constructed and was designed to gather the type of information that a revision system could use and that a user has the patience to provide, techniques learned in this research and new research could produce an industrial quality production system revision tool.

9.4 Final Thoughts

This dissertation has presented an approach to revising production system rulebases. A model of the revision problem space was defined and has successfully enabled a deeper understanding of the difficulties involved in revising production system rulebases. Most of these difficulties come about because only final fact-lists constraints are used to guide the revision process. As a research system, CR2 has been successful at illuminating many issues in this new and more complex area of theory revision. Future work in the field has the potential of turning what is a currently a research system into a practical industrial tool.

References

Anastasiadis, S. (1991). Rule groupings in expert systems using nearest neighbour decision rules, and convex hulls. In *Proceedings of the Second Conference on CLIPS*, (pp. 464–474).

Baffes, P. & Mooney, R. (1993). Symbolic revision of theories with m-of-n rules. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence, IJCAI-93*, (pp. 1135–1140).

Bergadano, F. & Giordana, A. (1988). A knowledge intensive approach to concept induction. In *Proceedings of the Fifth International Conference on Machine Learning, MLC88*, (pp. 305–317).

Brownston, L. (1985). *Programming expert systems in OPS5.* Reading, Mass: Addison-Wesley.

Brunk, C. & Pazzani, M. (1995). A linguistically-based semantic bias for theory revision. In *Machine Learning: Proceedings of the Twelfth International Conference*, (pp. 81–89).

Buchanan, B. & Shortliffe, E. (1984). *Rule-Based Expert Systems, The MYCIN Experiments of the Stanford Heuristic Programming Project.* Menlo Park, CA: Addison-Wesley Publishing Company.

Clancey, W. (1984). Classification problem solving. In *Proceedings of the Third National Conference on Artificial Intelligence, AAAI-84*, (pp. 49–55).

Davis, R. (1977). Interactive transfer of expertise: Acquisition of new inference rules. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence, IJCAI-77*, (pp. 321–328).

Davis, R. (1979). Interactive transfer of expertise: Acquisition of new inference rules. *Artificial Intelligence*, *12*(2), 121–157.

Giarratano, J. & Riley, G. (1993). *Expert Systems: principles and programming.* PWS Publishing Company, Boston, MA.

Ginsberg, A. (1988). Theory revision via prior operationalization. In *Proceedings of the Seventh National Conference on Artificial Intelligence, AAAI-88*, (pp. 590–595).

Ginsberg, A., Weiss, S., & Politakis, P. (1985). Seek2: A generalized approach to automatic knowledge base refinement. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence, IJCAI-85*, (pp. 367–374).

Grossner, C., Preece, A., Chander, P., Radhakrishnan, T., & Suen, C. (1993). Exploring the structure of rule based systems. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI93*, (pp. 704–709).

Jacob, R. & Froscher, J. (1990). A software engineering methodology for rule-based systems. *IEEE Transactions on Knowledge and Data Engineering*, *2*(2), 173–189.

Mehrotra, M. (1991). Rule groupings: An approach towards verification of expert systems. In *Proceedings of the Second Conference on CLIPS*, (pp. 21–24).

Morik, K. (1991). Balanced cooperative modeling. In *Proceedings of the First International Workshop on Multistrategy Learning*, (pp. 65–80).

Muggleton, S., Bain, M., Hayes-Michie, J., & Michie, D. (1989). An experimental comparison of human and machine learning formalisms. In *Proceedings of the Sixth International Workshop on Machine Learning*, (pp. 113–118).

Muggleton, S. & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Workshop on Machine Learning*, (pp. 339–352).

Muggleton, S. & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, (pp. 368–381).

Murphy, P. & Pazzani, M. (1994). Revision of production system rule-bases. In *Machine Learning: Proceedings of the Eleventh International Conference*, (pp. 199–207).

Ourston, D. (1991). *Using Explaination-Based and Empirical Methods in Theory Revision.* PhD thesis, University of Texas, Austin, Tx.

Park, Y. & Wilkin, D. (1990). Establishing the coherence of an explaination to improve refinement of an incomplete knowledge base. In *Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI90*, (pp. 511–516).

Pazzani, M.J., B. C. & Silverstein, G. (1991). A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning, MLW91*, (pp. 432–436).

Pazzani, M. & Brunk, C. (1991). *Detecting and Correcting Errors in Rule-Based Expert Systems: An Integration of Empirical and Explaination-Based Learning*, volume 3, (pp. 157–173).

Pazzani, M. & Kibler, D. (1992). The utility of knowledge in inductive learning. *Machine Learning, 9*(1), 57–94.

Politakis, P. & Weiss, S. (1984). Using empirical analysis to refine expert system knowledge bases. *Artificial Intelligence, 22*, 23–48.

Popplestone, R. (1970). An experiment in automatic induction. *Machine Intelligence, 5*, 203–215.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning, 5*, 239–266.

Richards, B. (1992). *An Operator-Based Approach to First-Order Theory Revision.* PhD thesis, University of Texas, Austin, Tx.

Tangkitvanich, S., N. M. & Shimura, M. (1992). Correcting multiple faults in the concept and subconcepts by learning and abduction. In *Proceedings of the International Workshop on Inductive Logic Programming.*

Tangkitvanich, S. & Shimura, M. (1993). Learning from an approximate theory and noisy examples. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI93*, (pp. 466–471).

Towell, G. (1991). *Symbolic knowledge and neural networks: Insertion, refinement and extraction.* PhD thesis, University of Wisconsin, Madison, WI.

Weiss, S. & Kulikowski, C. (1979). Expert: A system for developing consultation models. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence, IJCAI-79*, (pp. 942–947).

Wogulis, J. (1991). Revising relation domain theories. In *Proceedings of the Eighth International Workshop on Machine Learning, MLW91.*

Wogulis, J. (1994). *An Approach to Repairing and Evaluating First-Order Theories Containing Multiple Concepts and Negation.* PhD thesis, University of California, Irvine, CA.

Appendix A
Student Loan Domain

The student loan domain, in its original form, consists of a Horn clause rulebase and a set of instances that describe whether payment is due for a set of 1000 students who received student loans. The original rulebase is shown in Table A.1.

For this research, the Horn clause rulebase was converted to a production system rulebase by converting individual clauses into production rules. The LHS of the constructed production rule consisted of CEs formed from the literals in the clause's body. The RHS consisted of a single assert action that asserted the fact associated with the head of the clause. Each production rule had the same salience (zero). The constructed rulebase had 15 rules. No rules had retracts, printouts nor user query functions as actions.

One of the Horn clause rules, `enrolled_in_more_than_n_units/2`, could not be naturally converted into a production rule. For the two clauses that called `enrolled_in_more_than_n_units/2`, its body was used in place of the call when converting to a production rule. The production system version of this rulebase is shown in Table A.4.

Information encoded in the Horn clause predicates `school/1`, `armed_forces/1` and `peace_corps/1` was used by CR2 in the form of *deffacts* (facts that are asserted to the initial fact-list prior to running each instance). Table A.2 shows the *deffacts* for the student loan domain.

The instances used by CR2 were created directly from the original 1000 Horn clause based instances. Each CR2 instance consisted of a set of initial facts and a set of final fact-list constraints. The initial facts for an instance were formed from the ground facts associated with the Horn clause representation of the instance. The constraints on the final fact-list were constructed by asserting the initial facts to the fact-list and then running to completion the rules in the production system rulebase. An example instance is shown in Table A.3.

Table A.1. Student Loan production system rules

```
never_left_school(Student) :-
    longest_absence_from_school(Student,Units),
    6 > Units.
enrolled_in_more_than_n_units(Student,N) :-
    enrolled(Student,School,Units),
    school(School),
    Units > N.
no_payment_due(Student) :-
    continuously_enrolled(Student).
no_payment_due(Student) :-
    eligible_for_deferment(Student).
continuously_enrolled(Student) :-
    never_left_school(Student),
    enrolled_in_more_than_n_units(Student,5).
eligible_for_deferment(Student) :-
    military_deferment(Student).
eligible_for_deferment(Student) :-
    peace_corps_deferment(Student).
eligible_for_deferment(Student) :-
    financial_deferment(Student).
eligible_for_deferment(Student) :-
    student_deferment(Student).
eligible_for_deferment(Student) :-
    disability_deferment(Student).
military_deferment(Student) :-
    enlist(Student,Org),
    armed_forces(Org).
peace_corps_deferment(Student) :-
    enlist(Student,Org),
    peace_corps(Org).
financial_deferment(Student) :-
    filed_for_bankruptcy(Student).
financial_deferment(Student) :-
    unemployed(Student).
student_deferment(Student) :-
    enrolled_in_more_than_n_units(Student,11).
disability_deferment(Student) :-
    disabled(Student).
```

Table A.2. *deffacts* for the student loan domain.

- (school ucsd)
- (school ucb)
- (school ucla)
- (school uci)
- (school occ)
- (school smc)
- (armed-forces army)
- (armed-forces navy)
- (armed-forces air-force)
- (armed-forces marines)
- (peace-corps peace-corps)

Table A.3. Example instance from student loan domain.

```
Initial State Information:
   Initial Facts:
      (longest-absence-from-school 7)
      (enrolled ucsd 5)
      (enrolled occ 9)
      (enlist air-force)
      (unemployed)
Constraints on Execution of Rule-Base:
   Final Fact-list Constraints:
      Target: (no-payment-due)
      Others:
         (not (never-left-school))
         (not (continuously-enrolled))
         (eligible-for-deferment)
         (military-deferment)
         (not (peace-corps-deferment))
         (financial-deferment)
         (not (student-deferment))
         (not (disability-deferment))
```

Running the rulebase to completion caused the final fact-list to contain the deductive closure of the facts and rules in the rulebase. Positive final fact-list constraints were formed for each fact asserted by a rule to the final fact-list. Negative final fact-list constraints were formed from the assert action of each rule in the rulebase that did not fire. Duplicate negative final fact-list constraints were removed. Each of the 1000 instances ended up having a total of nine final fact-list constraints.

In order to avoid the possibility of infinite loops during rule execution, a limit of 20 rule executions per instances was used for this domain. No correct rule execution should fire any of the 15 rules more than once.

Table A.5 shows the four mutation version of the student loan rulebase that was used to produce some of the results for Chapter 8. The three mutation version used in that chapter looks like the rulebase in Table A.5, except that it does not have the missing clause mutation.

Table A.4. Student Loan production system rulebase.

```
(defrule never-left-school
    (longest-absence-from-school ?months)
    (test (> 6 ?months))
    =>
    (assert (never-left-school)))

(defrule no-payment-due1
    (continuously-enrolled)
    =>
    (assert (no-payment-due)))

(defrule no-payment-due2
    (eligible-for-deferment)
    =>
    (assert (no-payment-due)))

(defrule continuously-enrolled
    (never-left-school)
    (enrolled ?school ?units)
    (school ?school)
    (test (> ?units 5))
    =>
    (assert (continuously-enrolled)))

(defrule eligible-for-deferment1
    (military-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule eligible-for-deferment2
    (peace-corps-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule eligible-for-deferment3
    (financial-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule eligible-for-deferment4
    (student-deferment)
    =>
    (assert (eligible-for-deferment)))
```

```
(defrule eligible-for-deferment5
    (disability-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule military-deferment
    (enlist ?org)
    (armed-forces ?org)
    =>
    (assert (military-deferment)))

(defrule peace-corps-deferment
    (enlist ?org)
    (peace-corps ?org)
    =>
    (assert (peace-corps-deferment)))

(defrule financial-deferment1
    (filed-for-bankruptcy)
    =>
    (assert (financial-deferment)))

(defrule financial-deferment2
    (unemployed)
    =>
    (assert (financial-deferment)))

(defrule student-deferment
    (enrolled ?school ?units)
    (school ?school)
    (test (> ?units 11))
    =>
    (assert (student-deferment)))

(defrule disability-deferment
    (disabled)
    =>
    (assert (disability-deferment)))
```

Table A.5. Four mutation version of student-loan rulebase used in Chapter 8.

```
(defrule never-left-school
    (longest-absence-from-school ?months)
    (test (> 6 ?months))
    =>
    (assert (never-left-school)))

(defrule no-payment-due1
    (continuously-enrolled)
    =>
    (assert (no-payment-due)))

(defrule no-payment-due2
    (eligible-for-deferment)
    =>
    (assert (no-payment-due)))

(defrule continuously-enrolled
    % (never-left-school)              (missing CE)
    (enrolled ?school ?units)
    (school ?school)
    (test (> ?units 5))
    =>
    (assert (continuously-enrolled)))

(defrule eligible-for-deferment1
    (military-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule eligible-for-deferment2
    (peace-corps-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule eligible-for-deferment3
    (financial-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule eligible-for-deferment4
    (student-deferment)
    =>
    (assert (eligible-for-deferment)))
```

```
(defrule eligible-for-deferment5
    (disability-deferment)
    =>
    (assert (eligible-for-deferment)))

(defrule military-deferment
    (enlist ?org)
    (armed-forces ?org)
    =>
    (assert (military-deferment)))

(defrule peace-corps-deferment
    (enlist ?org)
    (peace-corps ?org)
    =>
    (assert (peace-corps-deferment)))

(defrule financial-deferment1
    (filed-for-bankruptcy)
    =>
    (assert (financial-deferment)))

%(defrule financial-deferment2          (missing rule)
%   (unemployed)
%   =>
%   (assert (financial-deferment)))

(defrule financial-deferment3           (extra rule)
    (enrolled uci ?units)
    =>
    (assert (financial-deferment)))

(defrule student-deferment
    (enrolled ?school ?units)
    (school ?school)
    (test (> ?units 11))
    =>
    (assert (student-deferment)))

(defrule disability-deferment
    (filed-for-bankrupcy)               (extra CE)
    (disabled)
    =>
    (assert (disability-deferment)))
```

Appendix B
Auto Diagnosis Domain

The auto diagnosis domain used in this research consists of a modified version of an example rulebase that is provided with the CLIPS distribution package. The purpose of the rulebase is to produce a single auto repair diagnosis, e.g. `(repair add-gas)`, based on the responses to a set of user query questions. There are 11 possible diagnoses.

The original version of the rulebase includes rules that have if-then-else actions (see Table B.1a). Since this research does not deal with if-then-else actions, each rule that includes such an action was converted to multiple rules that achieved the same purpose. The rules in Table B.1b were produced from the rule shown in Table B.1a. The rulebase shown in Table B.3 shows the 27 converted rules used in this research.

Note the partitioning of the rulebase into three sections. The first section, "Engine State Rules" corresponds to rules that performs inferences as data becomes available. These rules all have salience 10, one or more pattern conditional elements, and one or more assert actions. The second section, "User Query Rules", contains only rules that make user query function calls. Each rule has salience 0, a single bind (user query) and assert action and at least one negative pattern conditional element. The third section, "Miscellaneous Rules", contains all other rules. These rules tend to have very different purposes.

Unlike the student loan rulebase, which was originally a Horn clause rulebase, the auto diagnosis rulebase was designed to have structure. Many of the CLIPS rulebases that have been observed in the process of doing this research have structure. Figure B.1 shows the high-level structure and flow of rule execution for this rulebase.

In addition to structure, the auto diagnosis rulebase differs from the student-loan rulebase by the presence of multiple saliences, user query functions (bind actions), a "default classification rule", and the nature of rule execution. The default classification rule, no-repairs, fires when no other engine state rule or user query rule can fire.

Table B.1. Original and converted auto diagnosis rules

A. Original rule

```
(defrule determine-rotation-state
    (working-state engine does-not-start)
    (not (rotation-state engine ?))
    (not (repair ?))
    =>
    (if (yes-no-p Does-the-engine-rotate?)
      then
      (assert (rotation-state engine rotates))
      (assert (spark-state engine irregular-spark))
      else
      (assert (rotation-state engine does-not-rotate))
      (assert (spark-state engine does-not-spark))))
```

B. Converted rules

```
(defrule determine-rotation-state1
    (working-state engine does-not-start)
    (not (rotation-state engine ?))
    (not (repair ?))
    =>
    (bind ?response (ask-question how-does-engine-rotate?))
    (assert (rotation-state engine ?response)))

(defrule determine-rotation-state2
    (declare (salience 10))
    (working-state engine unsatisfactory)
    (rotation-state engine rotates)
    =>
    (assert (spark-state engine irregular-spark)))

(defrule determine-rotation-state3
    (declare (salience 10))
    (rotation-state engine does-not-rotate)
    =>
    (assert (spark-state engine does-not-spark)))
```

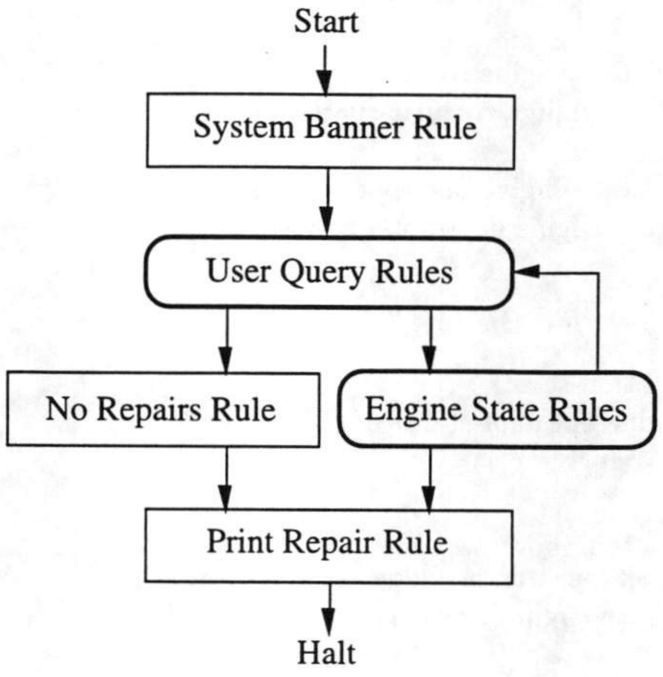


Figure B.1: Rulebase structure and flow of control for the auto diagnosis domain.

With respect to the nature of rule execution, the execution of the auto diagnosis rulebase depends not on a set of initial facts, but on the responses to user query function calls. Also, rulebase execution does not generate any kind of deductive closure, as is done for the student-loan rulebase. For the auto diagnosis rulebase, rule execution proceeds until the first diagnostic fact has been asserted to the fact-list, e.g., `(repair add-gas)`. Once a "repair" fact has been asserted, none of the "User Query Rules" can fire. Their firing is disabled by the presence of the negative pattern conditional element, `(not (repair ?))`.

No instances were provided with the original auto diagnosis rulebase. Therefore, a set of 250 instances were produced by randomly generating sets of answers to the user query questions. Most of these questions had either yes or no answers and no question had more than three possible answers. Instances were produced by running the rulebase while using each of the sets of answers. None of the instances included initial facts.

Constraints were formed using the contents of the final fact-list. Each of the facts in the final fact-list produced a positive final fact-list constraint. For each of the other possible diagnostic facts that were not asserted, negative final fact-list constraints were formed. For example, if the correct diagnosis was `(repair add-gas)`, then the negative final fact-list constraints would include `(not (repair charge-the-battery))`, `(not (repair no-repair-needed))` and `(not (repair clean-the-fuel-line))`. The number of final fact-list constraints varied between instances because negative final fact-list constraints were not produced for possible intermediate facts that were never asserted. Across all 250 instances, there are 4008 constraints, with no instance having fewer than 11 constraints. Table B.2 shows an example instance from the auto diagnosis domain.

The possibility of infinite loops during rule execution was handled by allowing a limit of 20 rule executions per instances. For this domain, no instance fires all 27 rules and no instances fires any rule multiple times.

Table B.2. Example instance from auto diagnosis domain.

```
Initial State Information:
  Initial Facts: NIL
  Function Call Bindings:
    (ask-question how-does-engine-run?) → does-not-start
    (yes-no-p how-does-engine-rotate?) → does-not-rotate
    (yes-no-p engine-sluggish?) → yes
    (yes-no-p does-engine-misfire?) → yes
    (yes-no-p engine-knocking?) → yes
    (yes-no-p engine-output?) → not-low-output
    (yes-no-p gas-in-tank?) → no
    (yes-no-p charge-state-battery?) → dead
    (yes-no-p surface-state-points?) → normal
    (yes-no-p conductivity-test-positive?) → no
Constraints on Execution of Rule-Base:
  Final Fact-list Constraints:
    Target: (repair charge-the-battery)
    Others:
      (not (repair no-repair-needed))
      (not (repair clean-the-fuel-line))
      (not (repair point-gap-adjustment))
      (not (repair timing-adjustment))
      (not (repair add-gas))
      (not (repair replace-the-points))
      (not (repair clean-the-points))
      (not (repair repair-the-distributor-lead-wire))
      (not (repair replace-the-ignition-coil))
      (not (repair take-your-car-to-a-mechanic))
      (working-state engine does-not-start)
      (rotation-state engine does-not-rotate)
      (spark-state engine does-not-spark)
      (charge-state battery dead)))
```

Table B.3. Auto Diagnosis rules

Engine State Rules

```
(defrule normal-engine-state-conclusions
    (declare (salience 10))
    (working-state engine normal)
    =>
    (assert (repair No-repair-needed))
    (assert (spark-state engine normal))
    (assert (charge-state battery charged))
    (assert (rotation-state engine rotates)))

(defrule unsatisfactory-engine-state-conclusions
    (declare (salience 10))
    (working-state engine unsatisfactory)
    =>
    (assert (charge-state battery charged))
    (assert (rotation-state engine rotates)))

(defrule determine-rotation-state2
    (declare (salience 10))
    (working-state engine unsatisfactory)
    (rotation-state engine rotates)
    =>
    (assert (spark-state engine irregular-spark)))

(defrule determine-rotation-state3
    (declare (salience 10))
    (rotation-state engine does-not-rotate)
    =>
    (assert (spark-state engine does-not-spark)))

(defrule determine-sluggishness2
    (declare (salience 10))
    (sluggish engine yes)
    =>
    (assert (repair Clean-the-fuel-line)))

(defrule determine-misfiring2
    (declare (salience 10))
    (misfire engine yes)
    =>
    (assert (repair Point-gap-adjustment))
    (assert (spark-state engine irregular-spark)))

(defrule determine-knocking2
    (declare (salience 10))
    (knocking engine yes)
    =>
    (assert (repair Timing-adjustment)))

(defrule determine-gas-level2
    (declare (salience 10))
    (gas-in-tank no)
    =>
    (assert (repair Add-gas)))

(defrule determine-battery-state2
    (declare (salience 10))
    (charge-state battery dead)
    =>
    (assert (repair Charge-the-battery)))

(defrule determine-point-surface-state3
    (declare (salience 10))
    (surface-state-points burned)
    =>
    (assert (repair Replace-the-points)))

(defrule determine-point-surface-state4
    (declare (salience 10))
    (surface-state-points contaminated)
    =>
    (assert (repair Clean-the-points)))

(defrule determine-conductivity-test2
    (declare (salience 10))
    (conductivity-test-positive yes)
    =>
    (assert (repair Repair-the-distributor-lead-wire)))

(defrule determine-conductivity-test3
    (declare (salience 10))
    (conductivity-test-positive no)
    =>
    (assert (repair Replace-the-ignition-coil)))
```

User Query Rules

```
(defrule determine-engine-state
    (not (working-state engine ?))
    (not (repair ?))
    =>
    (bind ?response (ask-question how-does-engine-run?))
    (assert (working-state engine ?response)))

(defrule determine-rotation-state1
    (working-state engine does-not-start)
    (not (rotation-state engine ?))
    (not (repair ?))
    =>
    (bind ?response (ask-question how-does-engine-rotate?))
    (assert (rotation-state engine ?response)))

(defrule determine-sluggishness1
    (working-state engine unsatisfactory)
    (not (repair ?))
    =>
    (bind ?response (ask-question engine-sluggish?))
    (assert (sluggish engine ?response)))

(defrule determine-misfiring1
    (working-state engine unsatisfactory)
    (not (repair ?))
    =>
    (bind ?response (ask-question does-engine-misfire?))
    (assert (misfire engine ?response)))

(defrule determine-knocking1
    (working-state engine unsatisfactory)
    (not (repair ?))
    =>
    (bind ?response (ask-question engine-knocking?))
    (assert (knocking engine ?response)))

(defrule determine-low-output
    (working-state engine unsatisfactory)
    (not (symptom engine low-output))
    (not (symptom engine not-low-output))
    (not (repair ?))
    =>
    (bind ?response (ask-question engine-output?))
    (assert (symptom engine ?response)))

(defrule determine-gas-level1
    (working-state engine does-not-start)
    (rotation-state engine rotates)
    (not (repair ?))
    =>
    (bind ?response (ask-question gas-in-tank?))
    (assert (gas-in-tank ?response)))

(defrule determine-battery-state1
    (rotation-state engine does-not-rotate)
    (not (charge-state battery ?))
    (not (repair ?))
    =>
    (bind ?response (ask-question charge-state-battery?))
    (assert (charge-state battery ?response)))

(defrule determine-point-surface-state1
    (working-state engine does-not-start)
    (spark-state engine irregular-spark)
    (not (repair ?))
    =>
    (bind ?response (ask-question surface-state-points?))
    (assert (surface-state-points ?response)))

(defrule determine-point-surface-state2
    (symptom engine low-output)
    (not (repair ?))
    =>
    (bind ?response (ask-question surface-state-points?))
    (assert (surface-state-points ?response)))

(defrule determine-conductivity-test1
    (working-state engine does-not-start)
    (spark-state engine does-not-spark)
    (charge-state battery charged)
    (not (repair ?))
    =>
    (bind ?response (ask-question conductivity-test-positive?))
    (assert (conductivity-test-positive ?response)))
```

Miscellaneous Rules

```
(defrule no-repairs
    (declare (salience -10))
    (not (repair ?))
    =>
    (assert (repair Take-your-car-to-a-mechanic)))

(defrule system-banner
    (declare (salience 10))
    =>
    (printout t The-Engine-Diagnosis-Expert-System))

(defrule print-repair
    (declare (salience 10))
    (repair ?item)
    =>
    (printout t Suggested-Repair)
    (printout t ?item))
```

Appendix C
Nematode Identification Domain

The nematode identification domain used in this research is a CLIPS rulebase that was created by a graduate student for an expert system course. The main purpose of the rulebase is to classify observed nematodes (Nematodes belong to the phylum *nematoda* and are sometimes referred to as roundworms), place the observed classifications into a simple database and manipulate database after all desired classification have been made. Twenty classes of nematoda are discriminatable by this rulebase.

The nematode rulebase used in this research is a slightly modified version of the original rulebase. In the original rulebase, user query rules included printout actions that presented the user with the question and possible answers, see Table C.1a. Query rules, used in this research, need to be in the form of a bind action that calls a user query function. The user query function uses the context of the argument passed as the question to determine which answer to return. Table C.1b shows a modified version of the rule in Table C.1a.

The nematode rulebase, shown in Table C.3, has 93 rules. Note the partitioning of the rulebase into sections. The purpose of this partitioning is to provide an overview of the subtasks performed by the rulebase. The rules in the first section, "Miscellaneous Initial Rules", perform a number of simple introductory functions including the display of the classes of nematode identifiable by the rulebase and the determination of whether the creature to be identified is a nematode. The second section, "Classification and Query Rules", includes rules that query the user and perform inferences on the results of responses to the query rules. The main purpose of the rules in this section is to classify observed nematodes. Multiple nematode identifications are accomplished by repeatedly executing the rules in this section. The third section, "Database Manipulation Rules", includes rules that are only executed after all classifications have been made. The rules in this section perform the tasks of printing the database and adding and deleting elements to the database.

Table C.1. Original and modified query rule

A. Original query rule

```
(defrule check-Trichorus
    (esophagus two-part)
    (not (Trichodorus ?answer))
    =>
    (printout t "Is this stylet short and curved, body short and thick "
                "(0.45-1.5 mm long) :" crlf
                " (yes/no) ? " crlf)
    (assert (Trichodorus =(read))))
```

B. Modified query rule

```
(defrule check-Trichorus
    (esophagus two-part)
    (not (Trichodorus ?answer))
    =>
    (bind ?response (yes-no-p stylet-short-curved?))
    (assert (Trichodorus ?response)))
```

The rules in this rulebase could also have been partitioned using structural characteristics. For example, many of the rules in this rulebase have a retract action for each CE. Most of the query rules in the "Classifications and Query Rules" section have a negative pattern CE and a single assert action that if executed would assert a fact that would match the negative pattern CE, e.g, not-Trichodorus and medium-bulb-size. Also, those rules that assert the actual classification of the observed nematode have exactly two assert actions, one for the classification and the other with a description of the reasons for the classification, e.g., Xiphinema and Longidorus.

In terms of similarities and differences between the nematode rulebase and the other rulebases, the nematode rulebase is most similar to the auto diagnosis rulebase. Both the nematode and the auto rulebases use user query functions instead of initial facts, have multiple saliences, and do not generate a deductive closure. The nematode identification rulebase is different from the auto diagnosis rulebase because it includes retract actions, is capable of making multiple classification and includes a procedural database manipulation component, to name a few.

No instances were originally provided with the nematode rulebase, so a set of 55 instances was produced by generating sets of answers to the user query questions. Because the answer returned by a user query function depends only on the question asked (and not on, for example, whether the question has been asked before), classification of multiple nematodes could not be tested in this research.

Instances were produced by running the rulebase while using each of the sets of answers. Constraints were formed using the contents of the final fact-list. Each of the facts in the final fact-list produced a positive final fact-list constraint. For all other possible nematode classifications that were not asserted, negative final fact-list constraints were formed. For most instances, no intermediate facts were available in the final fact-list to form positive final fact-list constraints. Table C.2 shows an example instance from the nematode identification domain.

The possibility of infinite loops during rule execution was avoided by placing a limit of 50 rule executions per instance.

Table C.2. Example instance from nematode identification domain.

```
Initial State Information:
  Initial Facts: NIL
  Function Call Bindings:
    (ask-question print-final-list?) → no
    (yes-no-p data-manipulation?) → add
    (yes-no-p print-nematode-identified-so-far?) → no
    (yes-no-p continue-id?) → no
    (yes-no-p stylet-extension-with-basal-flanges?) → no
    (yes-no-p stylet-long-straight-tapering?) → yes
    (yes-no-p stylet-short-curved?) → no
    (yes-no-p two-part-or-three-part?) → two-part
    (yes-no-p have-stylet?) → present
    (yes-no-p nematode-shape? → yes
    (yes-no-p nematode-p? → unknown
    (yes-no-p ready-to-work? → yes
    (yes-no-p which-nematode-to-add? → paralinda
    (yes-no-p more-to-modify? → no
Constraints on Execution of Rule-Base:
  Final Fact-list Constraints:
    (nema-id longidorus)
    (nema-id paralinda)
    (not (nema-id xiphinema))
    (not (nema-id tylenchulus))
    (not (nema-id tylenchorhynchus))
    (not (nema-id trichodorus))
    (not (nema-id rotylenchulus))
    (not (nema-id radopholus))
    (not (nema-id pratylenchus))
    (not (nema-id meloidodera))
    (not (nema-id meloidogyne))
    (not (nema-id hirshmanniella))
    (not (nema-id heterodera))
    (not (nema-id hemicriconemoides))
    (not (nema-id helicotylenchus))
    (not (nema-id ditylenchus))
    (not (nema-id criconema))
    (not (nema-id criconemoides))
    (not (nema-id aphelenchus))
    (not (nema-id aphelenchoides))
    (not (nema-id hoplolaimus))
    (not (nema-id "not included"))
    (not (nema-id "A large number of genera, feeding habits unknown."))
```

Table C.3. Nematode Identification rules

Miscellaneous Initial Rules

```
(defrule start
    (declare (salience 500))
    ?init < – (initial-fact)
    =>
    (printout t "Welcome to the expert nematode diagnosis system !")
    (printout t crlf)
    (printout t "This program can identify the following nematodes : ")
    (printout t crlf)
    (retract ?init)
    (assert (print-list list)))

(defrule print-list
    (declare (salience 500))
    (print-list list)
    ?genus < – (genus ?name)
    =>
    (retract ?genus)
    (printout t " Genus ")
    (printout t ?name)
    (printout t crlf))

(defrule ready
    ?print < – (print-list list)
    =>
    (retract ?print)
    (bind ?response (yes-no-p ready-to-work?))
    (assert (ready ?response)))

(defrule start-to-id
    ?ready < – (ready yes)
    =>
    (retract ?ready)
    (assert (query)))

(defrule determine-nematode
    ?query < – (query)
    (not (a nematode ?nema))
    =>
    (retract ?query)
    (bind ?response (ask-question nematode-p?))
    (assert (a nematode ?response)))

(defrule is-a-nematode-1
    ?f1 < − (a nematode yes)
    =>
    (retract ?f1)
    (assert (this-is-a-nematode)))

(defrule not-a-nema
    ?f1 < − (a nematode no)
    =>
    (retract ?f1)
    (bind ?response (yes-no-p not-nematode-find-another?))
    (assert (find-another ?response)))

(defrule find-another
    ?f4 < − (find-another yes)
    =>
    (assert (query))
    (retract ?f4))

(defrule unknown-creature
    (a nematode unknown)
    =>
    (bind ?response (yes-no-p nematode-shape?))
    (assert (valid-shape ?response)))

(defrule is-not-a-nema-2
    ?f1 < − (a nematode unknown)
    ?shape < − (valid-shape no)
    =>
    (bind ?response (yes-no-p not-nematode-find-another?))
    (assert (find-another ?response))
    (retract ?f1)
    (retract ?shape))

(defrule is-a-nema-2
    ?f1 < − (a nematode unknown)
    ?shape < − (valid-shape yes)
    =>
    (assert (this-is-a-nematode))
    (printout t "This is a nematode !")
    (printout t crlf)
    (retract ?f1)
    (retract ?shape))
```

Classification and Query Rules

```
(defrule stylet
    (this-is-a-nematode)
    (not (stylet ?present))
    =>
    (bind ?response (ask-question have-stylet?))
    (assert (stylet ?response)))

(defrule not-a-plant-parasitic-nema
    ?stylet < - (stylet absent)
    ?nema < - (this-is-a-nematode)
    =>
    (bind ?response (yes-no-p not-plant-parasitic-nematode-ready?))
    (assert (ready ?response))
    (retract ?stylet)
    (retract ?nema))

(defrule plant-parasitic-nema
    ?f1 < - (stylet present)
    ?f2 < - (this-is-a-nematode)
    (not (esophagus ?how-many-part))
    =>
    (retract ?f1)
    (retract ?f2)
    (bind ?response (ask-question two-part-or-three-part?))
    (assert (esophagus ?response)))

(defrule check-Trichorus
    (esophagus two-part)
    (not (Trichodorus ?answer))
    =>
    (bind ?response (yes-no-p stylet-short-curved?))
    (assert (Trichodorus ?response)))

(defrule Trichodorus
    ?f3 < - (esophagus two-part)
    ?f4 < - (Trichodorus yes)
    =>
    (retract ?f3)
    (retract ?f4)
    (assert (nematode Trichodorus))
    (assert (id-criteria "1. esophagus two part."
                        "2. body-shape short and thick."
                        "3. stylet-shape short and curved.")))

(defrule not-Trichodorus
    ?f2 < - (esophagus two-part)
    ?f1 < - (Trichodorus no)
    (not (Longidorus Xiphinema ?answer))
    =>
    (retract ?f1)
    (retract ?f2)
    (bind ?response (yes-no-p stylet-long-straight-tapering?))
    (assert (Longidorus Xiphinema ?response)))

(defrule Xiphinema-Logidorus
    (Longidorus Xiphinema yes)
    =>
    (bind ?response (yes-no-p stylet-extension-with-basal-flanges?))
    (assert (Xiphinema ?response)))

(defrule Xiphinema-facts
    ?f1 < - (Xiphinema yes)
    =>
    (retract ?f1)
    (assert (stylet-extension with basal-flanges))
    (assert (guiding-ring middle)))

(defrule Longidorus-facts
    ?f1 < - (Xiphinema no)
    =>
    (retract ?f1)
    (assert (stylet-extension without basal-flanges))
    (assert (guiding-ring anterior)))

(defrule Xiphinema
    ?f1 < - (Longidorus Xiphinema yes)
    ?f3 < - (stylet-extension with basal-flanges)
    ?f4 < - (guiding-ring middle)
    =>
    (retract ?f1)
    (retract ?f3)
    (retract ?f4)
    (assert (nematode Xiphinema))
    (assert (id-criteria "1. esophagus two-part."
                         "2. body-shape long and slender."
                         "3. stylet-shape long, straight, extension long ...")))

(defrule Longidorus
    ?f1 < – (Longidorus Xiphinema yes)
    ?f3 < – (stylet-extension without basal-flanges)
    ?f4 < – (guiding-ring anterior)
    =>
    (retract ?f1)
    (retract ?f3)
    (retract ?f4)
    (assert (nematode Longidorus))
    (assert (id-criteria "1. esophagus two-part."
                         "2. body-shape long and slender."
                         "3. stylet-shape long, straight, extension long ...")))

(defrule unknown-feeding-habits-nema
    ?f1 < – (Longidorus Xiphinema no)
    =>
    (retract ?f1)
    (assert (nematode "A large number of genera, feeding habits unknown."))
    (assert (id-criteria "1. esophagus two-part."
                         "2. stylet straight, usually not very long."
                         "3. body normal.")))

(defrule median-bulb-size
    (esophagus three-part)
    (not (median-bulb ?size))
    =>
    (bind ?response (ask-question size-median-bulb?))
    (assert (median-bulb ?response)))

(defrule metacorpus-small
    ?bulb < – (median-bulb 2)
    =>
    (retract ?bulb)
    (assert (median-bulb small)))

(defrule metacorpus-large
    ?bulb < – (median-bulb 1)
    =>
    (retract ?bulb)
    (assert (median-bulb large)))
```

```
(defrule Aphelenchoidea
    ?f1 < – (esophagus three-part)
    ?f2 < – (median-bulb large)
    (not (Superfamily Aphelenchoidea))
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (Superfamily Aphelenchoidea)))

(defrule Tylenchoidea
    ?f1 < – (esophagus three-part)
    ?f2 < – (median-bulb small)
    (not (Superfamily Tylenchoidea))
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (Superfamily Tylenchoidea)))

(defrule Aphelenchoidea-tail-shape
    (Superfamily Aphelenchoidea)
    (not (tail-shape ?size))
    =>
    (bind ?response (ask-question shape-tail?))
    (assert (tail-shape ?response)))

(defrule Aphelenchus
    ?f1 < – (Superfamily Aphelenchoidea)
    ?f2 < – (tail-shape blunt)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Aphelenchus))
    (assert (id-criteria "1. esophagus three-part."
                         "2. metacorpus large."
                         "3. female tail-shape blunt.")))

(defrule Aphelenchoides
    ?f1 < – (Superfamily Aphelenchoidea)
    ?f2 < – (tail-shape conoid)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Aphelenchoides))
    (assert (id-criteria "1. esophagus three-part."
                         "2. metacorpus large."
                         "3. tail-shape conoid, with 1 or more sharp points.")))

(defrule annulated-cuticle
    ?f1 < − (Superfamily Tylenchoidea)
    (not (cuticle-annulated-heavily ?any))
    =>
    (retract ?f1)
    (bind ?response (yes-no-p cuticle-heavily-annulated?))
    (assert (cuticle-annulated-heavily ?response)))

(defrule Criconematidae
    ?f2 < − (cuticle-annulated-heavily yes)
    (not (Family Criconematidae))
    =>
    (retract ?f2)
    (assert (Family Criconematidae)))

(defrule cuticle-sheath
    ?f3 < − (Family Criconematidae)
    (not (cuticle-sheath ?any))
    =>
    (retract ?f3)
    (bind ?response (ask-question body-prominent-cuticle-sheath?))
    (assert (cuticle-sheath ?response)))

(defrule Hemicriconemoides
    ?f4 < − (cuticle-sheath present)
    =>
    (retract ?f4)
    (assert (nematode Hemicriconemoides))
    (assert (id-criteria "1. esophagus three-part, metacorpus small."
                         "2. cuticle heavily annulated."
                         "3. cuticle-sheath prominent.")))

(defrule Criconema-Criconemoides
    ?f5 < − (cuticle-sheath absent)
    (not (annules-posterior-projections ?any))
    =>
    (retract ?f5)
    (bind ?response (yes-no-p annules-have-prominent-posterior-projections?))
    (assert (annules-posterior-projections ?response)))
```

```
(defrule Criconema
    ?f6 < - (annules-posterior-projections yes)
    =>
    (retract ?f6)
    (assert (nematode Criconema))
    (assert (id-criteria "1. esophagus three-part, metacorpus small."
                         "2. cuticle-sheath absent."
                         "3. cuticle-annules with posterior projections.")))

(defrule Criconemoides
    ?f7 < - (annules-posterior-projections no)
    =>
    (retract ?f7)
    (assert (nematode Criconemoides))
    (assert (id-criteria "1. esophagus three-part, metacorpus small."
                         "2. cuticle-sheath absent."
                         "3. heavy annulations, without posterior projections.")))

(defrule enlarge-body
    ?f8 < - (cuticle-annulated-heavily no)
    (not (female-body-shape ?any))
    =>
    (retract ?f8)
    (bind ?response (ask-question female-body-shape?))
    (assert (female-body-shape ?response)))

(defrule female-body-shape-1
    ?f1 < - (female-body-shape 1)
    =>
    (assert (female-body-shape pyriform-saccate-or-lemon-shape))
    (retract ?f1))

(defrule female-body-shape-2
    ?f2 < - (female-body-shape 2)
    =>
    (assert (female-body-shape elongate-saccate-or-kidney-shape-with-tail))
    (retract ?f2))

(defrule female-body-shape-3
    ?f3 < - (female-body-shape 3)
    =>
    (assert (female-body-shape cylindrical))
    (retract ?f3))
```

```
(defrule Heteroderidae
    ?f1 < − (female-body-shape pyriform-saccate-or-lemon-shape)
    (not (Family Heteroderidae))
    =>
    (retract ?f1)
    (assert (Family Heteroderidae)))

(defrule body-hardness
    (Family Heteroderidae)
    (not (female-body ?any))
    =>
    (bind ?response (ask-question female-body-hard-or-soft?))
    (assert (female-body ?response)))

(defrule Heterodera
    ?f1 < − (Family Heteroderidae)
    ?f2 < − (female-body hard)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Heterodera))
    (assert (id-criteria "1. esophagus three-part, metacorpus small."
                         "2. female-body lemon-shape."
                         "3. female-body hard-cyst.")))

(defrule Meloidodera-Meloidogyne
    (Family Heteroderidae)
    ?f1 < − (female-body soft)
    (not (vulva-position ?any))
    =>
    (retract ?f1)
    (bind ?response (ask-question position-of-vulva?))
    (assert (vulva-position ?response)))

(defrule vulva-position-terminal
    ?pos < − (vulva-position 1)
    =>
    (retract ?pos)
    (assert (vulva-position terminal)))

(defrule vulva-position-middle
    ?pos < − (vulva-position 2)
    =>
    (retract ?pos)
    (assert (vulva-position middle)))

(defrule Meloidogyne
    ?f1 < – (Family Heteroderidae)
    ?f2 < – (vulva-position terminal)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Meloidogyne))
    (assert (id-criteria "1. esophagus three-part, metacorpus small."
                         "2. female-body pear-shape, soft."
                         "3. vulva-position terminal of body.")))

(defrule Meloidodera
    ?f1 < – (Family Heteroderidae)
    ?f2 < – (vulva-position middle)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Meloidodera))
    (assert (id-criteria "1. esophagus three-part, metacorpus small."
                         "2. female-body pear-shape, soft."
                         "3. vulva-position middle of body.")))

(defrule esophagus-glands-intestine-1
    ?f1 < – (female-body-shape cylindrical)
    (not (esophagus-glands ?any))
    =>
    (retract ?f1)
    (bind ?response (ask-question where-esophagus-glands?))
    (assert (esophagus-glands ?response)))

(defrule esophagus-glands-intestine-2
    ?eso < – (esophagus-glands 1)
    =>
    (retract ?eso)
    (assert (esophagus-glands enclosed-in-basal-bulb)))

(defrule esophagus-glands-intestine-3
    ?eso < – (esophagus-glands 2)
    =>
    (retract ?eso)
    (assert (esophagus-glands overlaps-intestine)))
```

```
(defrule tail-shape-ovary
    (esophagus-glands enclosed-in-basal-bulb)
    (not (tail-shape ?any))
    =>
    (bind ?response (yes-no-p female-tail-blunt-rounded-two-ovaries?))
    (assert (tail-round ovary-two ?response)))

(defrule tail-ovary-2
    ?tail < - (tail-round ovary-two yes)
    =>
    (retract ?tail)
    (assert (tail-shape blunt-round yes))
    (assert (two-ovary yes)))

(defrule tail-ovary-3
    ?tail < - (tail-round ovary-two no)
    =>
    (retract ?tail)
    (assert (tail-shape blunt-round no))
    (assert (two-ovary no)))

(defrule Tylenchorhynchus
    ?f1 < - (esophagus-glands enclosed-in-basal-bulb)
    ?f2 < - (tail-shape blunt-round yes)
    ?f3 < - (two-ovary yes)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (assert (nematode Tylenchorhynchus))
    (assert (id-criteria "1. esophagus 3-part, metacorpus < 3/4 body width,
                         glands enclosed in basal-bulb."
                         "2. tail-shape blunt, rounded."
                         "3. ovary two.")))

(defrule Ditylenchus-Paratylenchus
    (esophagus-glands enclosed-in-basal-bulb)
    (tail-shape blunt-round no)
    (two-ovary no)
    (not (stylet-long ?any))
    =>
    (bind ?response (yes-no-p stylet-long?))
    (assert (stylet-long ?response)))
```

```
(defrule Ditylenchus
    ?f1 < – (esophagus-glands enclosed-in-basal-bulb)
    ?f2 < – (tail-shape blunt-round no)
    ?f3 < – (two-ovary no)
    ?f4 < – (stylet-long no)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (retract ?f4)
    (assert (nematode Ditylenchus))
    (assert (id-criteria "1. esophagus 3-part, glands enclosed in basal-bulb."
                         "2. tail-shape conoid with mucro."
                         "3. ovary one, outstretched")))

(defrule check-Paratylenchus
    (esophagus-glands enclosed-in-basal-bulb)
    (tail-shape blunt-round no)
    (two-ovary no)
    (stylet-long yes)
    (not (body-length-small ?any))
    =>
    (bind ?response (yes-no-p size-nematode-small?))
    (assert (body-length-small ?response)))

(defrule Paratylenchus
    ?f1 < – (esophagus-glands enclosed-in-basal-bulb)
    ?f2 < – (tail-shape blunt-round no)
    ?f3 < – (two-ovary no)
    ?f4 < – (stylet-long yes)
    ?f5 < – (body-length-small yes)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (retract ?f4)
    (retract ?f5)
    (assert (nematode Paratylenchus))
    (assert (id-criteria "1. esophagus three-part, metacorpus < 3/4 body width,
                         stylet 15 - 36 micron."
                         "2. tail-shape tapering, round at end."
                         "3. ovary one, body length < 0.5 mm")))
```

```
(defrule specimen-spiral
    (esophagus-glands overlaps-intestine)
    (not (specimen-spiral ?any))
    =>
    (bind ?response (yes-no-p specimen-lying-in-spiral?))
    (assert (specimen-spiral ?response)))

(defrule Helicotylenchus
    ?f1 < - (esophagus-glands overlaps-intestine)
    ?f2 < - (specimen-spiral yes)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Helicotylenchus))
    (assert (id-criteria "1. esophagus three-part, metacorpus < 3/4 body width."
                         "2. esophagus glands overlaps intestine."
                         "3. specimen lying in a spiral.")))

(defrule direction-overlaps
    ?f1 < - (esophagus-glands overlaps-intestine)
    ?f2 < - (specimen-spiral no)
    (not (esophagus-glands-overlap-intestine ?any))
    =>
    (retract ?f1)
    (retract ?f2)
    (bind ?response (ask-question where-overlap?))
    (assert (esophagus-glands-overlap-intestine ?response)))

(defrule Radopholus-Hoplolaimus
    (esophagus-glands-overlap-intestine dorsally)
    (not (female-head ?any))
    =>
    (bind ?response (ask-question female-head?))
    (assert (female-head ?response)))

(defrule head-shape-1
    ?head < - (female-head 1)
    =>
    (retract ?head)
    (assert (female-head flat)))

(defrule head-shape-2
    ?head < - (female-head 2)
    =>
    (retract ?head)
    (assert (female-head box-grid)))

(defrule check-Radopholus-1
    (female-head flat)
    (esophagus-glands-overlap-intestine dorsally)
    =>
    (bind ?response (ask-question how-many-ovaries?))
    (assert (ovary ?response)))

(defrule check-Radopholus-2
    ?f1 < – (female-head flat)
    ?f2 < – (esophagus-glands-overlap-intestine dorsally)
    ?f3 < – (ovary 2)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (assert (nematode Radopholus))
    (assert (id-criteria "1. esophagus three-part, metacorpus < 3/4 body width,
                         glands overlaps intestine dorsally."
                         "2. female-head low, rounded or flat."
                         "3. ovary two.")))

(defrule not-included-1
    ?f1 < – (female-head flat)
    ?f2 < – (esophagus-glands-overlap-intestine dorsally)
    ?f3 < – (ovary 1)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (assert (nematode "not included"))
    (assert (id-criteria "1. esophagus three-part,glands overlaps intestine ..."
                         "2. female-head low, rounded or flat."
                         "3. ovary 1.")))

(defrule Tylenchulus-Rotylenchulus
    (female-body-shape elongate-saccate-or-kidney-shape-with-tail)
    (not (ovary ?any))
    =>
    (bind ?response (ask-question how-many-ovaries?))
    (assert (ovary ?response)))
```

```
(defrule Tylenchulus
    ?f1 < − (female-body-shape elongate-saccate-or-kidney-shape-with-tail)
    ?f2 < − (ovary 1)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Tylenchulus))
    (assert (id-criteria "1. esophagus three-part, metacorpus < 3/4 body width."
                         "2. female-body-shape kidney-shape with tail."
                         "3. ovary one.")))

(defrule Rotylenchulus
    ?f1 < − (female-body-shape elongate-saccate-or-kidney-shape-with-tail)
    ?f2 < − (ovary 2)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Rotylenchulus))
    (assert (id-criteria "1. esophagus three-part, metacorpus < 3/4 body width."
                         "2. female-body elongate-saccate with tail."
                         "3. ovary two.")))

(defrule Pratylenchus-Hirshmanniella
    (esophagus-glands-overlap-intestine ventrally)
    (not (ovary ?any))
    =>
    (bind ?response (ask-question how-many-ovaries?))
    (assert (ovary ?response)))

(defrule Pratylenchus
    ?f1 < − (esophagus-glands-overlap-intestine ventrally)
    ?f2 < − (ovary 1)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Pratylenchus))
    (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                         "2. ovary 1."
                         "3. head-shape low and flat.")))

(defrule Hirshmanniella
    ?f1 < − (esophagus-glands-overlap-intestine ventrally)
    ?f2 < − (ovary 2)
    =>
    (retract ?f1)
    (retract ?f2)
    (assert (nematode Hirshmanniella))
    (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                         "2. ovary 2."
                         "3. head-shape low and flat.")))

(defrule check-Hoplolaiamus-1
    (esophagus-glands-overlap-intestine dorsally)
    (female-head box-grid)
    (not (spear-knob-with-projections ?any))
    =>
    (bind ?response (yes-no-p spear-have-knobs-prominent-anterior-projections?))
    (assert (spear-knob-with-projections ?response)))

(defrule not-included-2
    ?f1 < - (esophagus-glands-overlap-intestine dorsally)
    ?f2 < - (female-head box-grid)
    ?f3 < - (spear-knob-with-projections no)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (assert (nematode "not included"))
    (assert (id-criteria "1.esophagus 3-part."
                         "2. esophagus glands overlap intestine dorsally."
                         "3. spear knob without anterior projections.")))

(defrule Hoplolaimus
    ?f1 < - (esophagus-glands-overlap-intestine dorsally)
    ?f2 < - (female-head box-grid)
    ?f3 < - (spear-knob-with-projections yes)
    =>
    (retract ?f1)
    (retract ?f2)
    (retract ?f3)
    (assert (nematode Hoplolaimus))
    (assert (id-criteria "1. esophagus glands overlap intestine dorsally."
                         "2. female-head offset, caplike, divided into blocks by ..."
                         "3. spear knobs with anterior projections.")))

(defrule print-nematode
    (declare (salience -20))
    ?nema < – (nematode ?genus)
    =>
    (retract ?nema)
    (assert (nema ?genus))
    (printout t " Identification : ")
    (printout t " Genus ")
    (printout t ?genus)
    (printout t crlf)
    (printout t crlf))

(defrule print-characteristics
    (declare (salience -30))
    ?id < – (id-criteria ?fact1 ?fact2 ?fact3)
    =>
    (retract ?id)
    (printout t " Characteristics: ")
    (printout t ?fact1)
    (printout t crlf)
    (printout t " ")
    (printout t ?fact2)
    (printout t crlf)
    (printout t " ")
    (printout t ?fact3)
    (printout t crlf)
    (printout t crlf)
    (bind ?response (yes-no-p continue-id?))
    (assert (find-another ?response)))
```

Database Manipulation Rules

```
(defrule print-db-query
    ?f1 < – (find-another no)
    (nema ?genus)
    =>
    (retract ?f1)
    (bind ?response (yes-no-p print-nematode-identified-so-far?))
    (assert (print ?response)))

(defrule print-final-db-yes
    (declare (salience 10))
    ?print < – (print yes)
    ?nema < – (nema ?genus)
    =>
    (retract ?nema)
    (assert (nema-id ?genus))
    (printout t " Genus ")
    (printout t ?genus)
    (printout t crlf))

(defrule print-final-db-no
    (declare (salience 10))
    ?print < - (print no)
    ?nema < - (nema ?genus)
    =>
    (retract ?nema)
    (assert (nema-id ?genus)))

(defrule query-modification
    ?f1 < - (print ?any)
    (nema-id ?genus)
    =>
    (retract ?f1)
    (bind ?response (ask-question data-manipulation?))
    (assert (add-delete-search ?response)))

(defrule add-to-list
    ?f1 < - (add-delete-search add)
    =>
    (retract ?f1)
    (bind ?nema (ask-question which-nematode-to-add?))
    (assert (nema-id ?nema))
    (printout t " Genus " ?nema " is added to the list.")
    (printout t crlf)
    (bind ?response (ask-question more-to-modify?))
    (assert (add-delete-search ?response)))

(defrule delete-from-list-1
    ?f1 < - (add-delete-search delete)
    =>
    (retract ?f1)
    (bind ?response (ask-question which-nematode-to-delete?))
    (assert (delete ?response)))

(defrule delete-from-list-2
    ?f1 < - (delete ?nema)
    ?f2 < - (nema-id ?nema)
    =>
    (retract ?f1)
    (retract ?f2)
    (printout t "Genus ")
    (printout t ?nema)
    (printout t " is deleted from the list.")
    (printout t crlf)
    (bind ?response (ask-question more-to-modify?))
    (assert (add-delete-search ?response)))

(defrule search-database
    ?f1 < - (add-delete-search search)
    =>
    (retract ?f1)
    (bind ?response (ask-question which-nematode?))
    (assert (search ?response)))

(defrule data-found
    (declare (salience 10))
    ?search < - (search ?nema)
    (nema-id ?nema)
    =>
    (printout t "Genus ")
    (printout t ?nema)
    (printout t " is in the identified list.")
    (printout t crlf)
    (retract ?search)
    (bind ?response (ask-question more-to-modify?))
    (assert (add-delete-search ?response)))

(defrule data-not-found
    ?search < - (search ?nema)
    =>
    (printout t "Genus ")
    (printout t ?nema)
    (printout t " is not found in the identified list.")
    (printout t crlf)
    (retract ?search)
    (bind ?response (ask-question more-to-modify?))
    (assert (add-delete-search ?response)))

(defrule print-list-again-query
    ?f1 < - (add-delete-search no)
    =>
    (retract ?f1)
    (bind ?response (yes-no-p print-final-list?))
    (assert (print-again ?response)))

(defrule print-final-list
    (declare (salience 10))
    (print-again yes)
    ?nema < − (nema-id ?genus)
    =>
    (retract ?nema)
    (assert (nema-id ?genus))
    (printout t " Genus ")
    (printout t ?genus)
    (printout t crlf))
```

Appendix D
King-Rook-King Domain

The king-rook-king domain, in its original form, is a Horn clause rulebase that has been used by many researchers (Muggleton, Bain, Hayes-Michie & Michie, 1989; Muggleton & Feng, 1990; Pazzani & Kibler, 1992; Richards, 1992). The main task of the king-rook-king domain is to decide which chess board positions containing a white king, white rook and black king are illegal. A set of positions is considered illegal if the black king is in check or if any two pieces occupy the same board position. This domain is also sometimes referred to as "illegal".

The original Horn clause version of the king-rook-king rulebase could not be converted directly into a production system rulebase because many of the clauses in the Horn clause version had variables in their heads that were not bound in their bodies (see Table D.1).

A production rule formed from such a clause would have an assert for the head of the clause where not all variables are bound. Such rules are illegal for the production system language used in this research. Therefore, a new production system version of the king-rook-king rulebase was produced from scratch. The rulebase was produced in such a way that it encoded the same task as the original Horn clause version of the king-rook-king domain and also was easily encoded as a Horn clause rulebase. The new version has 15 rules and uses the same background information as the original Horn clause version.

A single mutated version of the king-rook-king domain was generated. This mutated rulebase had three missing CE mutations and two extra CE mutations in five rules. The production system version of this mutated rulebase is shown in Table D.4.

Two hundred instances were randomly generated from the correct king-rook-king rulebase. In addition to the target concept, `illegal`, each instance included three intermediate concepts. For CR2 the target concept became the target constraint and the intermediate concepts became the other constraints. The only initial fact associated with an instance was the position/6 fact which gave the position of each of the three pieces. An example CR2 king-rook-king instance

Table D.1. Original King-Rook-King Horn clause rulebase

```
krk(E) :- same_position(E).
krk(E) :- adj_kings(E).
krk(E) :- line_attack(E).
same_position(E) :- position(E,R,F,R,F,R3,F3).
same_position(E) :- position(E,R,F,R2,F2,R,F).
same_position(E) :- position(E,R1,F1,R,F,R,F).
adj_kings(E) :-
    position(E,R,F1,R2,F2,R,F3),
    adj(F1,F3).
adj_kings(E) :-
    position(E,R1,F,R2,F2,R3,F),
    adj(R1,R3).
adj_kings(E) :-
    position(E,R1,F1,R2,F2,R3,F3),
    adj(R1,R3),
    adj(F1,F3).
line_attack(E) :-
    position(E,R1,F1,R,F2,R,F3),
    not_equal(R1,R).
line_attack(E) :-
    position(E,R,F1,R,F2,R,F3),
    F1 < F2,
    F1 < F3.
line_attack(E) :-
    position(E,R,F1,R,F2,R,F3),
    F2 < F1,
    F3 < F1.
line_attack(E) :-
    position(E,R1,F1,R2,F,R3,F),
    not_equal(F1,F).
line_attack(E) :-
    position(E,R1,F,R2,F,R3,F),
    R1 < R2,
    R1 < R3.
line_attack(E) :-
    position(E,R1,F,R2,F,R3,F),
    R2 < R1,
    R3 < R1.
equal(X,X).
not_equal(X,Y) :- not(equal(X,Y)).
```

Table D.2. Example instance from king-rook-king domain.

```
Initial State Information:
  Initial Facts:
    (position 7 8 8 4 8 8)
Constraints on Execution of Rule-Base:
  Final Fact-list Constraints:
    Target: (illegal)
    Others:
      (adj-kings)
      (not (same-position))
      (line-attack)
```

is shown in Table D.2. For A3, each constraint from each instance produced an A3 example. The position/6 initial facts across all instances were used to form background information.

Facts that encode the extensional definition of the adj/2 concept were used by CR2 in the form of *deffacts*. Table D.3 shows the *deffacts* for the king-rook-king domain. For A3, these facts were used as background facts.

In order to avoid the possibility of infinite loops during rule execution, a limit of 50 rule executions per instances was used for this domain. No correct rule execution should fire any of the 15 rules more than once.

Table D.3. *deffacts* for the king-rook-king domain.

- (adj 1 2)
- (adj 2 3)
- (adj 3 4)
- (adj 4 5)
- (adj 5 6)
- (adj 6 7)
- (adj 7 8)
- (adj 8 7)
- (adj 7 6)
- (adj 6 5)
- (adj 5 4)
- (adj 4 3)
- (adj 3 2)
- (adj 2 1)

Table D.4. King-Rook-King production system rules

```
(defrule illegal-1
    (same-position)
    =>
    (assert (illegal)))

(defrule illegal-2
    (adj-kings)
    =>
    (assert (illegal)))

(defrule illegal-3
    (line-attack)
    =>
    (assert (illegal)))

(defrule same-position-1
    (position ?R ?F ?R ?F ?R3 ?F3)
    =>
    (assert (same-position)))

(defrule same-position-2
    (position ?R ?F ?R2 ?F2 ?R ?F)
    (test (= ?R ?R2))               (extra CE)
    =>
    (assert (same-position)))

(defrule same-position-3
    (position ?R1 ?F1 ?R ?F ?R ?F)
    =>
    (assert (same-position)))

(defrule adj-kings-1
    (position ?R ?F1 ?R2 ?F2 ?R ?F3)
    % (adj ?F1 ?F3)                 (missing CE)
    =>
    (assert (adj-kings)))

(defrule adj-kings-2
    (position ?R1 ?F ?R2 ?F2 ?R3 ?F)
    % (adj ?R1 ?R3)                 (missing CE)
    =>
    (assert (adj-kings)))
```

```
(defrule adj-kings-3
    (position ?R1 ?F1 ?R2 ?F2 ?R3 ?F3)
    % (adj ?R1 ?R3)                    (missing CE)
    (adj ?F1 ?F3)
    =>
    (assert (adj-kings)))

(defrule line-attack-1R
    (position ?R1 ?F1 ?R ?F2 ?R ?F3)
    (not (test (= ?R1 ?R)))
    =>
    (assert (line-attack)))

(defrule line-attack-2R
    (position ?R ?F1 ?R ?F2 ?R ?F3)
    (test (< ?F1 ?F2))
    (test (< ?F1 ?F3))
    =>
    (assert (line-attack)))

(defrule line-attack-3R
    (position ?R ?F1 ?R ?F2 ?R ?F3)
    (test (< ?F2 ?F1))
    (test (< ?F3 ?F1))
    =>
    (assert (line-attack)))

(defrule line-attack-1F
    (position ?R1 ?F1 ?R2 ?F ?R3 ?F)
    (not (test (= ?F1 ?F)))
    (test (= ?R1 ?R2))                 (extra CE)
    =>
    (assert (line-attack)))

(defrule line-attack-2F
    (position ?R1 ?F ?R2 ?F ?R3 ?F)
    (test (< ?R1 ?R2))
    (test (< ?R1 ?R3))
    =>
    (assert (line-attack)))
```

```
(defrule line-attack-3F
    (position ?R1 ?F ?R2 ?F ?R3 ?F)
    (test (< ?R2 ?R1))
    (test (< ?R3 ?R1))
    =>
    (assert (line-attack)))
```