Revision of Production System Rule-Bases

Patrick M. Murphy
Department of Information & Computer Science
University of California, Irvine, CA 92717
pmurphy@ics.uci.edu

Michael J. Pazzani
Department of Information & Computer Science
University of California, Irvine, CA 92717
pazzani@ics.uci.edu

Abstract

We describe CLIPS-R, a theory revision system for the revision of CLIPS rule-bases. CLIPS-R differs from previous theory revision systems in that it operates on forward chaining production systems. Revision of production system rule-bases is important because production systems can perform a variety of tasks such as monitoring and design in addition to classification tasks that have been addressed by previous research. We show that CLIPS-R can take advantage of a variety of user specified constraints on the correct processing of instances, such as ordering constraints on the displaying of information, and the contents of the final fact list. In addition, we show that CLIPS-R can operate as well as existing systems when the only constraint on processing an instance is the correct classification of the instance.

1 Introduction

Considerable progress has been made in the last few years in the subfield of machine learning known as theory revision (e.g. Ourston & Mooney, 1990; Pazzani & Brunk, 1991; Wogulis & Pazzani, 1993). The general goal of this area is to create learning models that can automatically update the knowledge base of a system to be more accurate on a set of test cases. Unfortunately, this progress has not yet been put into common practice. An important reason for the absence of technology transition is that only a restricted form of knowledge bases have been addressed. In particular, only the revision of logical knowledge bases that perform classification tasks with backward chaining rules (Clancey, 1984) has been explored. 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 that the production rule formalism is preferred to the logical rule 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, the production rule systems tend to be computationally more efficient. The production systems allow the knowledge engineer to have more influence over the flow of control in the systems allowing the performance to be fine tuned. That is, in a logical 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.

The revision of CLIPS rule-bases presents a number of challenging problems that have not been addressed in previous research on theory revision. In particular, rules can retract facts from working memory, display information, and request user input. New opportunities to take advantage of additional sources of information also accompany these new problems. For example, a user might provide information that a certain item that was displayed should not have been, or that information is displayed in the wrong order.

In the remainder of this paper, we first give a quick overview of CLIPS, and then describe a system CLIPS-R for revising CLIPS rule-bases. We evaluate CLIPS-R on the student loan problem, that we have translated from PROLOG into CLIPS to show that CLIPS-R is competitive with existing theory revision systems. In addition, we present results obtained by inserting random errors into a sample CLIPS rule-base that uses CLIPS features such as displaying information and querying the user.

CLIPS-R is still in its infancy and we expect many of the details of the individual operators and heuristics to change as this work matures.

2 The CLIPS Production System Language

The CLIPS production system language was created and is currently maintained by NASA for use in numerous knowledge-based application. In addition to NASA, its base of users include thousands of industrial and educational sites (NASA, 1991). In this section, we describe a subset of the most recent version of CLIPS (version 6.0), that should be sufficient to follow our examples.

CLIPS is similar to a variety of production system languages, in that it has a rule-base, an agenda (an ordered sequence of rule activations), and a fact-list (working memory). 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.

The position within the agenda that a new rule activation is placed is a function of the rule's salience and the current conflict resolution strategy. In our experiments, we use the Breadth Strategy in which an activation is placed above all activations of lower salience and below all activation of equal or higher salience.

```
(defrule determine-point-surface-state
   (declare (salience 10))
   (or (and (working-state engine does-not-start)))
            (spark-state engine irregular-spark))
       (symptom engine low-output))
   (not (repair ?))
   =>
   (bind ?response
       (ask-question What-is-surface-state-of-the-pnts?))
   (if (eq ?response burned)
       then (assert (repair Replace-the-points))
       else (if (eq ?response contaminated)
               then (assert (repair Clean-the-points)))))
```

(a) Example rule from Auto Diagnosis Rule-base

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

(b) Example rule from Student Loan Rule-base

Figure 1. Example CLIPS Rules

Figure 1 shows some examples of CLIPS rules[1]. A rule has an optional initial declaration section, which for the rule in Figure 1a declares that the rule's salience[2] is 10, a left-hand side (LHS) or antecedent of the rule and a right-hand side (RHS) or the consequent of the rule which follows =>. The LHS consists of a set of conditional elements that each must be satisfied to form a rule activation. The two primitive conditional elements dealt with in this research are the pattern conditional element and the test conditional element. Conditional elements may also be arbitrary Boolean expressions of primitive conditional elements. Pattern conditional elements are satisfied by matching a fact in the fact-list. They are represented as a sequence of constants and variables. Test conditional elements correspond to variable-variable or variable-constant comparisons[3], such as = and > .

To satisfy the first conditional element for the rule in Figure 1a, either the facts *(working-state engine does-not-start)* and *(spark-state engine irregular-spark)* must both be in the fact-list or the fact *(symptom engine low-output)* must be in the fact-list[4]. To satisfy the second conditional element, *(not (repair ?))*, there can be no fact of length 2 in the fact-list whose first element is repair. The second conditional element in Figure 1b, *(test (> 6 ?Units))*, is a test conditional element. For this rule to be satisfied, there must be a two-parameter fact in the fact-list with first parameter *longest-absence-from-school* and second parameter a number less than 6.

The RHS of a rule consists of an order sequence of actions[5]. The two most common actions, assert and retract, modify the fact-list. The assert action adds ground facts to the fact-list. The retract action removes a fact from the fact-list. The retract in the rule of Figure 2 retracts the fact that matched the conditional element *(phase choose-qualities)*.

```
(defrule change-to-phase-2
   (declare (salience -10))
   ?phase <-  (phase choose-qualities)
   =>
   (retract ?phase)
   (assert (phase recommend-qualities)))
```

Figure 2. Example of Retract Action

Other actions are if-then-else control structures and various functions. Variables may be introduced and bound within the action section of a rule using the bind action. In Figure 1a, the result of calling the function ask-question with parameter *What-is-surface-state-of-pnts?*, is stored in the variable *?response*. The function ask-question, prints the question to the screen and waits for a response. If the test for the if-then-else control structure evaluates to true the sequence of actions in the then part are executed, otherwise the actions in the else part are executed.

[1]Variables in CLIPS rules are represented by a question mark followed by a symbol, e.g. *?response*. Variables represented using a single question mark are unique for each occurrence in the rule and are similar to anonymous variables in PROLOG.

[2]When omitted, the salience of a rule defaults to 0.

[3]The variables come from pattern conditions elements.

[4]If each of these three facts were present in the fact-list, and the second conditional element was also satisfied, two rule activations would be formed for this rule.

[5]Some actions are functions in that they return a value, while other actions serve only to create a side effect.

3 Theory Revision

Most logical theory revision systems take as input a theory and a set of instances that are not fully consistent with the theory. The instances are classified tuples or sets of propositions and the theory is usually a set of horn clauses. Theory revision systems typically have an iterative refinement control structure that includes: identification of a set of likely repairs (usually done through an abductive process) and evaluation of the repairs over the set of instances. The repair that produces the greatest decrease in the error is retained and the process repeats until either all instances are correctly classified or no change results in a decrease in error. The repairs consist mainly of logical generalization and specialization operations on existing rules and the induction of new rules.

Like logical theory revision systems, CLIPS-R has an iterative refinement control structure (see Figure 3), and operators for specialization and generalization of existing rules. However, CLIPS-R has a more generalized notion of an instance, a novel method for identifying which repairs should be attempted and a larger set of repair operators.

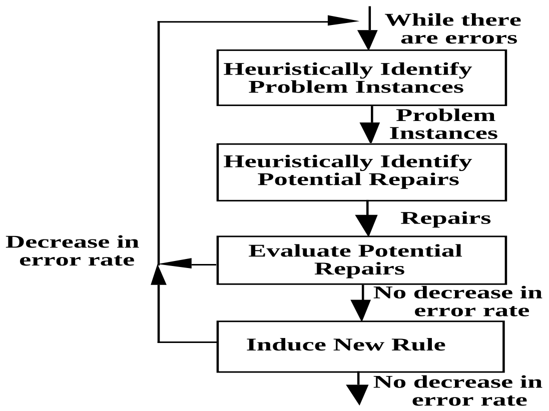

While there are errors
Heuristically Identify Problem Instances
Problem Instances
Heuristically Identify Potential Repairs
Repairs
Decrease in error rate
Evaluate Potential Repairs
No decrease in error rate
Induce New Rule
No decrease in error rate

Figure 3. System Organization

3.1 Instance Description

An instance used by CLIPS-R has two components: initial state information and constraints on the execution of the rule-base given the initial state. The initial state information consists of a set of initial facts to be loaded into the fact-list before execution of the rule-base, and a set of bindings that relates a function call (represented by a function name and values for its arguments) to its return value[6]. The set of constraints on the execution of the rule-base includes constraints on the contents of the final fact-list[7] and constraints on the ordering of observable actions such as displaying data or asking the user questions.

[6]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. These bindings, recorded by interactively running CLIPS, are saved and later used to avoid asking the user questions during theory revision.

[7]The final fact-list is the fact-list when execution of the rule-base halts.

Final fact-list constraint violations 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, for the constraint *(not (working-state engine normal))* to be satisfied, it should not match any fact in the final fact-list. Constraints on the ordering of the execution of observable actions, are represented using a finite state machine. Ordering constraints are satisfied when the sequence of observable actions, formed from the execution of an instance, is accepted by the finite state machine associated with that instance. Observable actions are user identified.

3.2 Instance Evaluation

To evaluate an instance with respect to its constraints, it is first executed while recording specific trace information. Unlike most theory revision systems where individual instances are given a score of 0 or 1 depending on whether they are correctly or incorrectly classified by the theory, individual instances in CLIPS-R are each given an error score that ranges from 0 to 1. The error of an individual instance is simply the number of constraint violations divided by the total number of possible constraint violations. The error score for multiple instances is the average of their individual error scores.

Initial State Information:
  Initial Facts: (none)
  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 Rule-Base:
  Final Fact-list Constraints:
    Target: (repair timing-adjustment)
    Others: (none)
  Ordering Constraint:

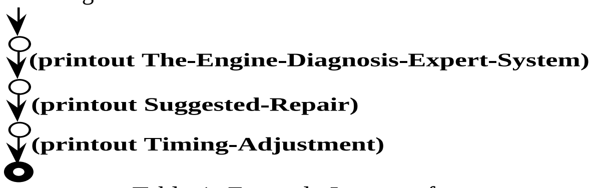

(printout The-Engine-Diagnosis-Expert-System)
(printout Suggested-Repair)
(printout Timing-Adjustment)

Table 1. Example Instance from Auto Diagnosis Rule-Base

The number of final fact-list constraint violations is just the total number of unsatisfied final fact-list constraints. All final fact-list constraint violations are equally weighted. The number of ordering constraint violations is found by determining the minimum number of additions and deletions required to be made to the observable action sequence such that the sequence is accepted by the finite state machine. Table 1 shows an example instance from the auto diagnosis rule-base.

3.3 Identifying Which Revision to Consider

CLIPS-R attempts to fix the most important errors first. In order to achieve this, it runs the current rule-base on each instance and maintains a trace of rule firings. The sequences of rule firings are used to form a trie structure (Fredkin, 1960). The trie structure is a compact representation of all of the rule traces that groups together those instances that share an initial sequence of rule firings. Each leaf stores the set of instances with the same sequence of rule firings. Associated with each instance is its individual error rate as well as an augmented trace of rule firings (see section 3.6). In addition, associated with each internal node of the trie is a sum of the number of errors of all instances with rule firing sequences that share that node and a sum of the total number of possible errors. An error rate for each node is obtained by dividing the total number of errors at the node by the total number of possible errors at the node. Figure 4 shows an example trie that summarizes five instances processed using the incorrect student loan rule-base. Nine constraints were evaluated for each instances, and the three numbers (e.g. [6,45,0.133]) associated with each node are the total number of errors, the total possible number of errors and the error rate, respectively, at that node.

```
Rule: continuously-enrolled [6,45,0.133]
  Rule: student-deferment [6,45,0.133]
    Rule: no-payment-due [5,27,0.185]
      Rule: eligible-for-deferment [1,9,0.111]
        Rule: no-payment-due [1,9,0.111]
          Leaf: [1,9,0.111]
      Leaf: [4,18,0.222]
    Rule: never-left-school [1,18,0.056]
      Rule: financial-deferment [1,9,0.111]
        Rule: no-payment-due [1,9,0.111]
          Rule: eligible-for-deferment [1,9,0.111]
            Rule: eligible-for-deferment [1,9,0.111]
              Rule: no-payment-due [1,9,0.111]
                Leaf: [1,9,0.111]
      Leaf: [0,9,0.000]
```

Figure 4. Example Trie for Incorrect Student Loan Rule-Base Created from Five Instances

The trie structure is used to identify sets of instances with relatively large numbers of constraint violations and that have a common sequence of rule firings. To do this, a search from root to leaf is done, where at each step, the internal node with highest error rate[8] is chosen. The product of this search is a leaf of instances associated with a single sequence of rule firings. In Figure 4, the underlined nodes corresponds to the worst path through the trie. The leaf at the end of this path contains two instances with a total of four constraint violations.

3.4 Identifying Which Revision to Make

For each constraint violation of each instance at the worst path leaf, a set of repair identification heuristics are used to generate a set of candidate repairs (a candidate repair is a call to a repair operator). Each candidate repair is then evaluated on the complete set of instances and the repair that produces the greatest decrease in average error rate over all of the instances is made. We will first describe some of the repair operators used by CLIPS-R and then discuss the heuristics that suggest which repairs to evaluate.

3.5 Operators

In addition to rule specialization and generalization operations typically found in previous theory revision systems, CLIPS-R has a number of other operators for dealing with CLIPS's rich rule representation language. Each operator typically generates several possible revisions, each of which is evaluated, with the best revisions retained. Each operator takes as input the rule to be modified and other revision specific information. Because of space limitations, not all operators will be fully defined.

3.5.1 LHS Specialization & Generalization

In order to simplify the description of the specialization and generalization operators, it is useful to conceptualize the set of conditional elements in a rule as a single Boolean expression. Given this representation, specialization is accomplished by either conjoining a primitive conditional element to a term in the expression or by deleting a disjunct from the expression. Generalization is accomplished by either disjoining a primitive conditional element to a term in the expression or by deleting a conjunct from the expression.

Pattern conditional element are added by considering sequences of variables and constants. The allowable patterns come from an automated analysis of the patterns found in the initial rule-base and the training examples. Given the set of all patterns extracted from the initial rule-base and training examples, an exhaustive set of pattern templates[9] is generated, such that each template uniquely matches a subset of the extracted patterns. The set of allowable patterns selected for evaluation is formed by variablizing[10] each of the pattern templates in all possible ways.

Test conditional elements are added by considering all combinations of a comparison operator and either two variables or a variable and a constant. The constants that can be com-

[8]Ties are broken by choosing the node with highest total number of possible errors.

[9]A pattern template is syntactically equivalent to a pattern conditional element.

[10]Variables may either be new or may come from previous pattern conditional elements.

pared to a variable are determined by collecting the set of values that may be bound to that variable. For example in Figure 1b, the possible values for the variable *?Units* is determined by looking at the second parameter of all facts whose first element is *longest-absence-from-school*.

3.5.2 Action Promotion & Demotion

The purpose of the action promotion and demotion operators are to decrease and increase, respectively, the number of conditions that must be satisfied before an action within a fired rule is executed. For example in Figure 5, the demotion operator could move an action at position A to position B while the promotion operator could move an action at position B to position A.

```
(defrule determine-engine-state
   (not (working-state engine ?))
   (not (repair ?))
   =>
   (assert (working-state engine does-not-start)) ;;; A
   (if (yes-no-p Does-the-engine-start?)
       then
       (if (yes-no-p Does-the-engine-run-normally?)
           then (assert (working-state engine normal))
           else (assert (working-state engine unsatisfactory)))
       else
       (assert (working-state engine does-not-start)))) ;;; B
```

Figure 5. Example of Promotion & Demotion Operator

The demotion operator also has the ability to embed the action in a new if-then-else control structure. The action is moved to either the then part or the else part of the new if-then-else control structure and the new if-then-else control structure is moved to the position once held by the demoted action. User provided templates for allowable test functions for new if-then-else control structures are used to construct the test functions. The template defines type and value constraints on the arguments of the function. The best if-then-else control structure is selected by evaluating all template-based instantiations of the test function. Lastly, the demotion operator considers deletion of the action.

3.5.3 Assert & Retract Addition

The assert and retract addition operators allow fired rules to assert and retract facts that should and should not, respectively, be present in the fact-list when the rule was fired. The add assert operator takes as input a pattern that should match any fact asserted by the added action. Revariablizations, using existing variables, of the pattern form the set of assert actions that are evaluated. The add retract action takes as input a pattern that should match the fact retracted by the added action. Retract actions are formed by taking each pattern conditional element in the LHS of the rule that is subsumed by the pattern, associating a reference variable to it, and using that reference variable as argument to the new retract action. When adding a new assert or retract action, only unordered positions in the RHS of the rule are considered, e.g. in Figure 7, positions *A*, *B* and *D*, only.

```
(defrule student-deferment
   =>
   ;;; Position A
   (if (enrolled-in-more-than-n-units 11)
       then
         ;;; Position B
         (assert (student-deferment))
         ;;; Position C
       else
         ;;; Position D)
   ;;; Position E)
```

Figure 7. Example of the positions where an action may be inserted.

3.5.4 Observable Action Modification

The operators for fixing violations of observable actions are similar to those for fixing incorrect assertions. For example, if an observable was detected that should not have been (or observed out of sequence) the action that created that observable is considered for demotion or deletion. When adding an action that creates an observable, all possible ordered positions within the RHS of the rule are considered, e.g. in Figure 7, positions *A*, *B*, *C*, *D* and *E*.

3.5.5 Changing Rule Salience

The salience of a rule may be increased to make the rule fire ahead of other rules that are satisfied. All possible distinguishable increases in salience are attempted if an effect of the RHS of the rule should have occurred but did not. Similarly, the salience may be decreased if an effect of the RHS occurred, but should not have occurred.

3.5.6 Rule Deletion

A rule deletion operator exists for removing rules that execute actions that should not have been executed.

3.6 Repair Identification Heuristics

Repair identification heuristics are used to suggest a set of operators that should be attempted to improve the rule-base. There are two sets of heuristics that may be used: final fact-list heuristics suggest how to correct errors when the assertions (e.g., the class of the instance) made when processing an instance, do not agree with the desired assertions. Ordering constraint violation heuristics suggest how to correct errors when observables are omitted or occur in the wrong order. If both kinds of violations occur, then both kinds of heuristics suggest repairs and the best repair is made.

Repairs are suggested to fix only those constraint violations of instances at the worst leaf in the trie structure. A set of functions exist for identifying repairs for certain types of constraint violations. The information used by these functions include a detailed trace of the rule firings for the instance with the constraint violation, and other information associated with the actual constraint violation.

The trace information includes:

- The sequence of rule firings.
- A copy of the fact-list prior to each rule firing.
- A copy of the final fact-list.
- Lists of actions executed by each rule firing.
- Information associating a fact and its source (i.e. an assert action).
- Information associating a retract action and the fact that it retracted.
- Information associating the positive pattern conditional elements of each fired rule and the facts that matched them.

Other information includes the specific fact that causes an extra-fact constraint violation, or the pattern, associated to a missing condition, that should have but did not match any fact in the final fact-list.

3.6.1 Final Fact-list Constraint Violation Repair Identification

Constraint violations on the final fact-list are in the form of extra ground facts and missing conditions (i.e., a general pattern that should unify with a ground fact in the fact-list). Below is a description of the set of repair identification heuristics for these two types of problems. In addition, some of the repair operators invoke other classes of repair operators (Extra Rule Firing and Missing Rule firing) to suggest repairs.

Extra Fact

- The action within the rule that asserted this fact should be demoted.
- Promote any unexecuted retract actions in a previously fired rule that could have retracted the fact.
- Add a retract for the fact to a previously fired rule.
- For each unfired rule in the rule-base that has a retract that could retract this fact, identify repairs that would allow that rule to fire (See Missing Rule Firing).
- Identify repairs that could cause the rule that asserted the fact to not fire (see Extra Rule Firing).

Missing Condition

- Promote unexecuted assert actions in previously fired rules that could have asserted a fact matching the condition.
- Add an assert for the condition to a previously fired rule.
- Demote any retract actions in fired rules that retracted facts that matched the condition.
- For each unfired rule in the rule-base that has an action that could have asserted a fact that matches the condition, identify repairs that could allow the rule to fire (See Missing Rule Firing).
- Identify repairs that could make any fired rule that retracted a fact that matched the condition, not fire (see Extra Rule Firing).

Extra Rule Firing

- Specialize LHS of rule.
- Delete rule.
- Decrease Salience of rule.
- For each satisfied unnegated conditional element in the rule, identify repairs that could cause the fact to not have been present in the fact-list at the time that the rule fired (See Extra Fact).
- For each satisfied negated conditional element in the rule, identify repairs that could cause some fact to be in the fact-list at the time that the rule fired (See Missing Condition).

Missing Rule Firing

- Generalize LHS of rule.
- Increase Salience of rule.
- For each unsatisfied unnegated conditional element in the rule, identify repairs that could allow a fact to have been present in the fact-list at the time that the rule should have fired (See Missing Condition).
- For each unsatisfied negated conditional element in the rule, identify repairs that could cause all facts that caused the conditional element to be unsatisfied to be missing from the fact-list (See Extra Fact).

For example, a final fact-list constraint violations, caused by an extra fact, could be repaired by demoting the assert action of the rule that asserted the fact, or by making the rule that asserted the extra fact not fire (e.g. specialize the rule). Repair identification heuristics are recursive. For example, another method for stopping the above mentioned rule from firing would be to consider one of the facts that matched the LHS of the rule as an extra fact in the fact-list at the time that the rule fired.

3.6.2 Ordering Constraint Violation Repair Identification

Constraint violations on the ordering of a sequence of observables are repaired by identifying observable actions that should be added to or deleted from the sequence. When the deletion of an action from the sequence is one of the identified repairs, a repair to demote (which includes delete) that action from the firing rule is selected for evaluation. When the addition of an action to the sequence is an identified repair, a set of repairs for adding that action to each rule fired between the last and next correctly matched observable action are selected for evaluation.

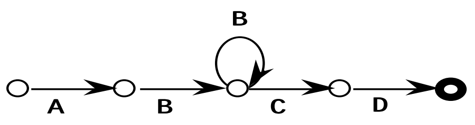


Figure 8. Example finite state machine.

For example, assume that the actions A, B, C and D are identified by the user as observable actions. The sequence of observable actions A, C, C, D would be accepted by the finite state machine in Figure 8 if an action B was executed between actions A and C and if the second C action was not executed. All minimal sets of repairs, for this example, are of size two.

If the sequence of rule firings was R1, R2, R3, R4, and R5, with rule R1 executing action A, R4 executing action C, and R5 executing actions C and than D, e.g.

$$R1(A) \rightarrow R2() \rightarrow R3() \rightarrow R4(C) \rightarrow R5(C,D),$$

the repairs suggested by the heuristics would be:

- Demote action C from rule R5 and
- Add action B to rule R1, or
- Add action B to rule R2, or
- Add action B to rule R3, or
- Add action B to rule R4.

An alternative minimal set of repairs would similarly include the addition of action B between actions A and C, but instead would include the deletion of the first C action. The new repairs suggested by the heuristics would be:

- Demote action C from rule R4 and
- Add action B to rule R5.

All seven of the above repairs would be identified for evaluation.

Note that the sets of repairs that would include the addition of multiple B actions and the deletion of a C action, while being accepted by the finite state machine, would not be a minimal set of repairs and would not identify repairs for evaluation.

3.7 Rule Induction

When other repair operators fail, an attempt is made to induce a new rule to solve some final fact- list constraint violation. A sampling of rules are first generated then, in turn, revised through a local hill-climbing iterative refinement search. The rule with the lowest error when added to the rule-base is added. Initial rules are generated in the following manner. The LHS of the rule is formed by taking the least general generalization (Plotkin, 1970) of the initial fact-lists for a pair of instances that have similar final fact-list constraint violations. If the common constraint violation corresponds to a positive constraint (e.g. a missing condition in the final fact-list) the RHS of the rule contains a single assert of the missing condition. Similarly, if the final fact-list constraint violation corresponds to a negative constraint (e.g. an extra fact in the final fact-list) the RHS contains a single retract for the extra fact.

4 Experiments

A series of experiments were designed to analyze various characteristics of CLIPS-R. In the first domain, we show that CLIPS-R is competitive with other theory revision systems on the student loan rule-base (Pazzani & Brunk, 1991), a problem translated from PROLOG to CLIPS. In addition, on this domain, we show that CLIPS can take advantage of information that FOCL-FRONTIER (Pazzani & Brunk, 1993) cannot by including information on whether some intermediate conclusions are true of each instance. In the second domain, we deal with the auto diagnosis rule-base that is distributed with CLIPS. This experiment uses many of the features of CLIPS including saliences, displaying data, and querying the user for information.

4.1 Student Loan Rule-Base

The student loan domain consists of a set of nine rules (represented as Horn clauses) and a set of 1000 instances. The rule-base contains four errors (an extra literal, a missing literal, an extra clause and a missing clause). This initial theory has an error of 21.6%. In order to use this rule-base with CLIPS-R, the nine Horn clause rules were converted into nine production rules, each with a single assert action. Multiple clauses in the Horn clause rules were converted to a disjunction of conjuncts within a CLIPS production rule.

Execution of an instance for the student loan rule-base consisted of asserting into an empty fact-list a set of facts specific to the instance and then executing the rule-base to completion[11]. All results for the following experiments are averages of 20 runs. All instances not used for training are used for testing.

Train Size	FOCL % Error	CLIPS-R % Error
25	11.8	12.6
50	5.8	3.0
75	2.8	2.1

Table 2. A Comparison of FOCL and CLIPS-R

The first experiment performed was to determine how well CLIPS-R performed at revising the rule-base relative to FOCL-FRONTIER. Table 2 shows that the error rate is competitive with that of FOCL-FRONTIER on this problem. Only with 50 training examples is the difference in error significant ($p < .05$).

The second experiment performed using this rule-base was to determine if CLIPS-R could take advantage of intermediate concepts during revision. In the first experiment, CLIPS-R was only given a single constraint (on the target concept), either *(no-payment-due)* or *(not (no-payment-due))*. For this experiment, CLIPS-R is also given varying percentages of the eight intermediate concepts as constraints, e.g. *(eligible-for-military-deferment)* or *(not (eligible-for-military-deferment))*. For each instance and percentage of intermediate concepts, a random subset of the intermediate concepts was selected to form the constraints.

[11] A rule count execution limit, provided as input to CLIPS-R, stops execution of an instance if the number of rules executed is greater than the limit. When the limit is reached, the maximum error possible is returned as the evaluation for the instance.

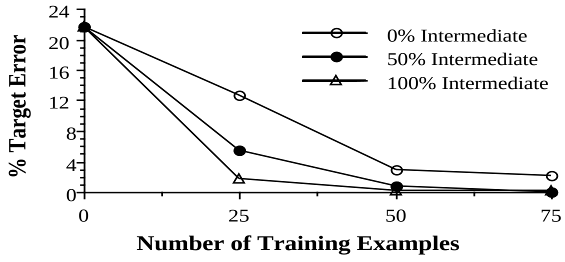


Figure 9. Percent Target Error as a function of Training Set Size when using 0%, 50% and 100% of the Intermediate Concepts during Revision.

Figure 9 shows percent error on the target concept as a function of training set size for three percentages of intermediate concepts. As percentage of intermediate concepts increases, error on the target concept decreases. In fact, the error rate at training set size 25 with 100% of the intermediate concepts is less than the error rate at training set size 75 with no intermediate concepts.

4.2 Auto Diagnosis Rule-Base

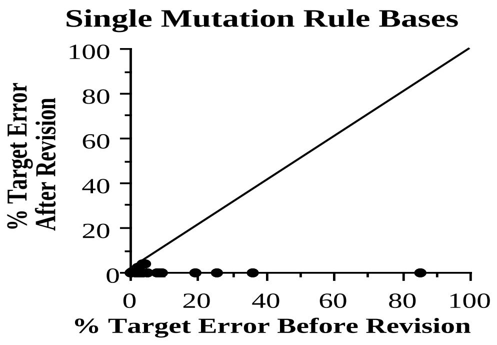
The auto diagnosis rule-base (provided with the CLIPS software) uses features of CLIPS not easily expressed in pure Horn clauses. It is an expert system that prints out an introductory message, asks a series of questions of the user, and prints out a concluding message including the predicted diagnosis. This rule-base has a total of 15 rules.

Instances were generated from 258 combinations of responses to the user query function. Each instance consisted of a set of answers for each invocation of the query function (e.g., see Figure 1a) as initial state information, a single constraint on the final fact-list and an ordering constraint for the sequence of printout actions. The target constraint for each instance was a positive constraint for a repair fact, e.g. *(repair Replace-the-points)*.

Execution of an instance for the auto diagnosis rule-base consisted of clearing the fact-list, setting the bindings that determine the return values for each function call instance (to simulate user input for the user query function) and executing the rule-base to completion. The bindings that associate function calls to their return values allowed an otherwise interactive rule-base to be run in batch mode. This is necessary because no user would be willing to answer the same questions for 50 instances on different variations of the rule-base.

The experiments performed using the auto diagnosis rule-base were designed to determine how well CLIPS-R could do at revising mutated versions of the correct rule-base. Mutations consisted of extra, missing or incorrect[12] conditional elements or actions and incorrect rule salience values. Two sets of 20 mutated rule-bases were randomly generated with one set of rule-bases having only a single mutation and the other set having three mutations per rule-base. Each mutated rule-base was revised using a random set of 50 training instances. The remaining instances were used for testing. Figure 10 contains a scatter plot showing the initial error of each mutated rule-base and the final error after revision of the rule-base.

[12] An incorrect conditional element was formed by changing the last argument of the conditional element to some other possible value. Wrong assert actions were formed similarly.



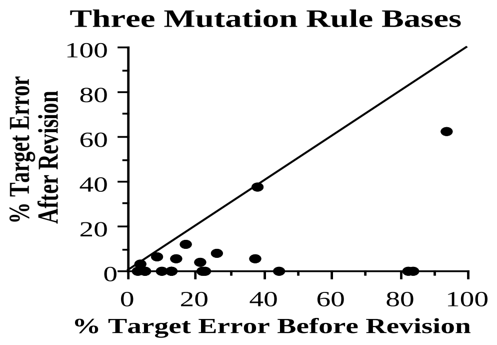


Figure 10. Target Error After Revision as a function of Target Error Before Revision.

An analysis of the scatter plots in Figure 10 shows that, for the most part, CLIPS-R was able to reduce the error rates of the mutated rule-bases (points below the diagonal indicate a decrease in target error). For one mutation, the average rule-base error was 11.2% before learning and 0.7% after learning. With three mutations, the error before learning was 28.0% and after learning it was 7.2%.

5 Related Work

Most of the previous work with production systems concerned cognitive modeling, e.g. PSG (Newell & McDermott, 1975), OPS (Forgy & McDermott, 1976) and PRISM (Langley & Neches, 1981). Some, including PRISM, have dealt with learning and revision of knowledge-bases. Most of the models explored using these systems have been very different from the CLIPS production system model. For example, some systems had multiple working memories, e.g. ACTE (Anderson, 1976), other systems associated strengths with production rules that, when combined with the conflict resolution strategy, make the firing of strong rules more likely. Previous work on learning included generalization and discrimination (specialization) of rules, e.g.

PRISM (Langley, 1987), and composition of rules, e.g. Lewis (1978). CLIPS-R differs from this prior research in a variety of ways including its focus on batch processing of a set of instances to find minimal revisions to a rule-base created by a knowledge engineer, its strategies to dealing with the complex RHS of CLIPS productions, its approach for identifying which rules to consider repairing, and its use of a variety of constraints on the correct processing of an instance.

6 Conclusion

We have described CLIPS-R, a theory revision system for the revision of CLIPS rule-bases. Novel aspects of CLIPS-R include the ability to handle forward chaining theories with "nonlogical" operations such as rule saliences and the retraction of information from working memory. We have introduced an approach to focusing the revision process of a forward chaining production system on the changes that are likely to be most beneficial. CLIPS-R can take advantage of a variety of user specified constraints on the correct processing of instances such as ordering constraints on the displaying of information, and the contents of the final fact-list. In addition, we have shown that CLIPS-R can operate as well as existing systems when the only constraint on processing an instance is the correct classification of the instance.

Acknowledgments

The research reported here was supported in part by NSF Grant IRI-9310413, ARPA Grant F49620-92-J-0430, and AFOSR AASERT grant F49620-93-1-0569.

References

Anderson, J. R. (1978). *Language, Memory, and Thought.* Lawrence Erlbaum Associates, Hillsdale, N.J.

Clancey, W. (1984). Classification problem solving. *Proceedings of the National Conference on Artificial Intelligence.* Morgan Kaufmann, Austin, TX, 49-55.

Forgy, C. L. & McDermott, J. (1976). *The OPS reference manual.* Technical Report. Department of Computer Science, Carnegie-Mellon University.

Fredkin, E. H. (1960). Trie Memory. *Communications of the ACM*, 3:9, 490-500.

Langley, P. & Neches, R. T. (1981). *PRISM users manual.* Technical Report. Computer Science Department, Carnegie-Mellon University.

Langley, P. (1987). A General Theory of Discrimination Learning. *In Production System Models of Learning and Development*, MIT Press, Cambridge Massachusetts, 99-161.

Lewis, C. (1978). *Production system models of practice effects.* Dissertation. Department of Psychology, University of Michigan.

NASA: Software Technology Branch. (1991) *CLIPS Reference Manual, Basic Programming Guide*, Volume 1, CLIPS Version 5.1. Lyndon B. Johnson Space Center.

Newell, A. & McDermott, J. (1975). *PSG manual.* Technical Report. Department of Computer Science, Carnegie-Mellon University.

Ourston, D. & Mooney, R. (1990). Changing the rules: A comprehensive approach to theory refinement. *Proceedings of the Eighth National Conference on Artificial Intelligence*. Boston, MA: Morgan Kaufmann, 815-820.

Pazzani, M., & Brunk, C. (1991). Detecting and correcting errors in rule-based expert systems: an integration of empirical and explanation-based learning. *Knowledge Acquisition*, 3, 157-173.

Pazzani, M. & Brunk, C. (1993). Finding Accurate Frontiers: A Knowledge-Intensive Approach to Relational Learning. *Proceedings of the Eleventh National Conference on Artificial Intelligence*. Washington, D.C, 328-334.

Pazzani, M., & Kibler, D. (1992). The Utility of Knowledge in Inductive Learning. *Machine Learning*, 9:1, 57-94.

Plotkin, G. (1970). A Note on Inductive Generalization. *In Machine Intelligence*, Vol. 5, Edinburgh University Press, Edinburgh, 153-163.

Wogulis, J. & Pazzani, M. (1993). A methodology for evaluating theory revision systems: Results with AUDREY II. *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, Chambery, France.