A METALINGUISTIC APPROACH TO THE CONSTRUCTION OF KNOWLEDGE BASE REFINEMENT SYSTEMS

Allen Ginsberg

AT&T Bell Laboratories
Holmdel, NJ 07733

Abstract

A variety of approaches to knowledge base refinement [3, 8] and rule acquistion [4] have appeared recently. This paper is concerned with the means by which alternative refinement systems themselves may be specified, developed, and studied. The anticipated virtues of taking a metalinguistic approach to these tasks are described, and shown to be confirmed by experience with an actual refinement metalanguage, RM.

I Introduction

Knowledge base refinement involves the generation, testing, and possible incorporation of *plausible refinements* to the rules in a knowledge base with the intention of thereby improving its empirical adequacy, i.e., its ability to correctly diagnose or classify the *cases* in its "domain of expertise." Knowledge base refinement may thus be viewed as being a part of, or a well-constrained subcase of, the knowledge acquisition problem*.

Recently a variety of methods or approaches to rule refinement for expert system knowledge bases have been presented [3, 8]. In [3] my colleagues and I discussed SEEK2, a system utilizing an *empirically-grounded heuristic* approach. In [8] an approach that makes use of explicit *rule justification structures* based upon the underlying domain theory was presented. An approach to rule acquistion using interview strategies based upon metaknowledge concerning the role of qualitative or causal models in diagnostic reasoning is given in [4]. The concern of this paper, however, is not with any particular refinement system or approach as such, but rather with the means by which refinement systems *themselves* may be designed, implemented, refined, and studied. In this paper a metalinguistic framework for accomplishing these tasks, called RM (for Refinement Metalanguage), will be described. RM is a metalanguage within which one may specify a wide variety of *rule* refinement systems.

In section II a brief elaboration on the general idea of a metalanguage for knowledge base refinement is given. Section III deals with the reasons for taking the "metalinguistic turn." Sections IV presents some of the salient features and primitives of RM, and gives examples of their use. Some concrete results concerning RM's performance are reported in section V.

II The Metalinguistic Turn

First we discuss the role of a rule refinement system within the context of the traditional expert or rule-based systems paradigm. A knowledge base is a collection of rules written in some *rule representation language*. In addition, an expert system framework contains an *inference mechanism* for determining if and how satisfied rules will be used to reach conclusions for any given case. By comparing the conclusions of a given knowledge base in one or more cases with the known conclusions in those cases, a decision can be made (either by a knowledge engineer or the refinement system) as to the current need for rule refinement. When invoked, the refinement system will - at the very least - suggest ways of modifying the rules in the knowledge base that are either *likely* to correct the given performance deficiencies or, as in the case of SEEK2 [3], have been *verified* to yield a certain specific gain in performance. From this perspective a refinement system is a useful "black box."

Figure II-1 illustrates what is meant by taking "the metalinguistic turn."

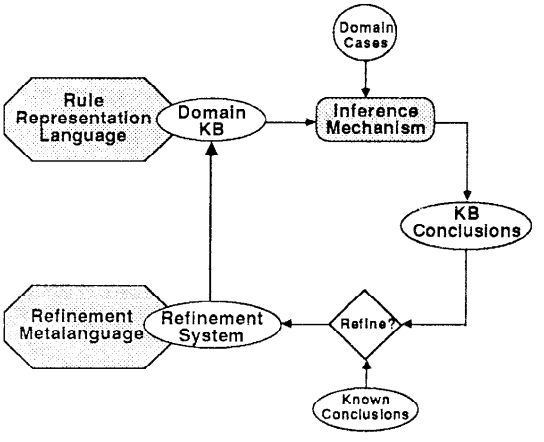

Figure II-1: Refinement Systems: After the Metalinguistic Turn

In this figure we see that the refinement system *itself* is viewed as a formal object written in a general metalanguage. Just as the rule representation language allows for the specification of many different knowledge bases, so too the refinement metalanguage allows for the specification of many different refinement systems for refining knowledge bases written in the rule representation language. Using the terminology of formal logic, we say that the refinement metalanguage is a *meta*-language with respect to the rule representation *object-language* because the former must have the ability to refer to, examine, and modify the linguistic entities that are definable in the latter. The metalinguistic turn may be viewed as running parallel to, or extending, the well-known generalization process that led from special purpose hard-coded expert systems to general rule representation frameworks or languages.

The metalanguage presented in this paper, RM, is currently capable of dealing with object-level rules that are expressible in an *extended propositional logic*, where the latter is a propositional logic that allows for confidence factors, and certain special forms - e.g., numerical ranges. In concrete terms, RM is designed for use with knowledge bases written in EXPERT [9].

*This research was conducted at the Department of Computer Science of Rutgers University and was supported in part by the Division of Research Resources, National Institutes of Health, Public Health Service, Department of Health, Education, and Welfare, Grant P41 RR02230.

III Motivation

Why is the "metalinguistic turn" worth taking? To put the question another way, what perceived needs are met or advantages realized by using a refinement metalanguage to design, implement, and study refinement systems?

One answer is in fact presupposed by the question itself: there is an expectation that the design and development of refinement systems is a field that is by no means a closed book. A prime motivation for creating a refinement metalanguage is, therefore, to have a tool for facilitating experimental research in knowledge base refinement. A system that allows for the easy specification of alternative refinement concepts, heuristics, and strategies, is *ipso facto*, a system that is useful for testing and comparing alternative refinement systems designs.

Moreover, as we have found in developing SEEK2 [3, 2], a refinement system undergoes debugging, revision, and expansion over time. Such development and evolution is much more easily understood, and hence, managed and achieved, by means of a high-level metalanguage, such as RM, than by use of traditional programming techniques. Again the reasoning here parallels the reasoning behind the evolution of expert system *frameworks* from hard-coded systems. Just as an expert system framework allows a knowledge engineer to concentrate on the essential problem at hand, viz., extracting knowledge from a domain expert, without worrying about irrelevant programming issues, so too a refinement metalanguage allows a researcher or refinement system designer to concentrate on the discovery and use of *metaknowledge* for facilitating knowledge base refinement, without being concerned about *how* such metaknowledge is to be represented and incorporated in a computer program.

A second reason for taking the metalinguistic turn is that there is no reason to believe that there is one "best" refinement system or overall approach to knowledge base refinement. Applicable and useful refinement strategies and tactics may vary from domain to domain, and may also be a function of the stage of development of the object knowledge base. The metalinguistic approach to the specification of refinement systems allows for the easy customization of refinement systems to meet these varying circumstances.

A closely related motivation concerns the use of *domain-specific metaknowledge* in refinement systems, i.e., metaknowledge concerning the particular linguistic entities in some domain knowledge base. Falling in this category, for example, would be the metaknowledge that certain rules are "known with certainty" and should not be refined under any circumstances. A more interesting example of such metaknowledge is the notion of a *generalization language* [6] or *causal network* [10] defined over the concepts utilized by the rules of the knowledge base. An example of how such domain-specific metaknowledge can be utilized will be discussed in section IV.B.2. The point to be made here is that, while domain-specific metaknowledge concerns, by definition, a particular domain knowledge base, such metaknowledge can be represented by general metalinguistic devices. Therefore a refinement metalanguage is a natural vehicle for the representation and incorporation of such metaknowledge in the refinement process.

Finally, note that the metalinguistic turn allows us to contemplate the implementation of a refinement system that is capable of refining itself, i.e., given the accessibility of its own metalinguistic representation there is no reason in principle why a refinement system should not be able to refine its own concepts, heuristics, and procedures in order to improve its performance [5]. For example, if a refinement system Σ is given feedback, i.e., told which of its suggested refinements are unacceptable, Σ may be able to use this information together with the record of its reasoning in these instances to suggest refinements to the concepts, heuristics, or procedures that led to these erroneous suggestions.

IV Salient Features of RM

A. Primitives for Accessing Object-Language Entities and Properties

In order to allow for the specification of a wide variety of control strategies for the knowledge base refinement process [2], a refinement metalanguage must have the ability to create, store, and manipulate refined versions of the initial or given knowledge base.

There are many ways in which such a capability can be implemented. For the sake of efficiency, RM currently employs a scheme in which a distinction is maintained between the potentially numerous versions of the knowledge base that are *accessible*, and the *unique* version that is currently *active*. In RM **kb** is a primitive variable over the set of all accessible refined versions of the knowledge base; $\mathbf{kb_0}$ is a *constant* that refers to the initial knowledge base. Below we discuss the primitives provided by RM for creating and accessing refined versions of knowledge base, and some of the implementation details (section IV.B.6). Part of what it means for a knowledge base to be *the active* knowledge base will be made clear in the following paragraph.

A refinement metalanguage must possess primitive variables and functions to provide it (or its user) with the ability to "access" various "objects" or their components in *any* of the available versions of the knowledge base (as well as any of the domain cases that may be stored). In RM, variables referring to knowledge base entities are always interpreted as referring to the objects in *the active knowledge base*. For example, **rule** is a primitive variable whose range is the set of rules in the active knowledge base; **RuleCF(rule)** is a function whose value is the confidence factor associated with **rule** in the active knowledge base. To use these primitives to access or refer to rules, etc., in any accessible **kb** *other than* the active knowledge base, **kb** must first be *activated*. This is accomplished in RM by issuing the command **Activate-kb(kb)**.

RM views a *domain case* as a complex object consisting of a fixed collection of data or findings (including the known or expert conclusions in the case), i.e., the content of a case should not be subject to alteration by a refinement system. **case** is a variable whose range is the set of cases in the data base of cases. In addition some primitive *functions* are needed to allow one to refer to selected parts or aspects of a rule or a case, e.g., **PDX(case)** is a function whose value is the *known or expert conclusion* in **case** ("PDX" stands for "Presumed Diagnosis"); **Value(finding,case)**, is a function that returns the value of **finding** in **case**.

CDX(case) is a function whose value is the conclusion reached (with highest confidence) by the knowledge base in **case** ("CDX" stands for "Computer's Diagnosis"). **CDX-Total(case)** returns a vector containing all diagnostic or interpretive conclusions reached in case together with their confidence. While the content of any case is fixed (as far as RM is concerned), the conclusions reached by a various refined versions of $\mathbf{kb_0}$ in a case will, of course, vary. Therefore, the value returned by **CDX(case)** depends upon which knowledge base is active.

Some primitives can be used to return information concerning either rules, rule components, or subcomponents of rule components. This is achieved by a system for referencing such objects, the details of which need not concern us here. Basically, to refer to a rule, one pointer is required; to refer to a rule component, two pointers are required, etc. For example the RM primitive function **Satisfied** can be invoked in the following ways:

Satisfied(rule,case)

Satisfied(rule,component,case)

Satisfied(rule,component,subcomponent,case)

and will return *true* if and only if the designated rule, rule component, or rule subcomponent is indeed satisfied in the designated case.

Certain special sets of objects are of importance in the knowledge base refinement process, and it is therefore useful to have primitives that refer to them, e.g., **Rules-For(hypothesis)** is a function whose value is the set of rules that have **hypothesis** as their conclusion (**hypothesis** is, of course, a primitive variable ranging over the set of hypotheses in the knowledge base). In addition, it is desirable to have the ability to refer to *subsets* of various objects. Thus, in RM, **cases** is a primitive variable over subsets of cases, **rules** is a primitive variable over subsets of rules.

Other primitives that in some way involve *semantic* properties of rules, or the *performance characteristics* of the knowledge base as a whole are clearly required. For example, **ModelCF(hypothesis,case)** is a function whose value is the system's confidence factor accorded to **hypothesis** in **case**.

B. Primitives for Defining Functions, Modifying the Knowledge Base, and Creating Objects

1. Logical Operators

A refinement metalanguage must provide operators for combining primitive functions in order to form sophisticated functions. The basic operators that are needed are familiar from set theory, logic, and arithmetic. Some procedural or algorithmic primitives are needed as well. Since RM is built "on top" of common lisp, most of the primitive operations needed are already available as lisp primitives, and may be used in RM commands and definitions. Examples of these operators will be given in the course of the exposition.

2. Modifying the Knowledge Base

A metalanguage for rule refinement must provide an adequate set of *primitive rule refinement operators*. It should allow for these operators to be composed, so that *higher-order* refinement is possible [2]. The primitive refinement operations currently available in RM are ones that we have found to be useful in developing SEEK2 [3]. However, RM offers a set of primitive operators that is not only sufficient for the definition of SEEK2, but also goes beyond SEEK2 in power. For example, in SEEK2 all refinement operators apply only to rule components (including confidence factors); there is no way SEEK2 can apply an operator to a rule subcomponent. In RM, on the other hand, rule subcomponents can be accessed and manipulated in the same manner as top-level components.

As an example, the expression

(operation delete-component rule component)

denotes the operation of deleting **component** from **rule**. The expression

```
(operation
  (operation delete-component rule component)
  (operation lower-cf rule x))
```

denotes the *compound* operation of deleting **component** from **rule** and simultaneously lowering **rule**'s confidence factor to the new value **x**.

Refinement operators such as **delete-component** and **lower-cf** are specifiable in a purely syntactic manner. Refinement operators that are specifiable *semantically* may also, however, be incorporated in a refinement system. As an example, suppose that a *generalization language* [6] for concepts used in the rules is provided along with the knowledge base. Then *concept-generalization* and *concept-specialization* can be made available as primitive refinement operations. In RM the operation

(operation generalize-concept rule component)

replaces the item designated by the two arguments **rule, component** with the next "more general" item in the generalization language. As a concrete example of such an operation, a rule component corresponding to the finding **(Patient has a swollen ankle)**, might be generalized to **(Patient has a swollen joint)**.

3. Creating Mathematical Objects

A refinement metalanguage must give a user the ability to define sophisticated functions using the primitives. In order for this to be possible a user must be able to *create* new sets, relations, functions, and new variables (over both individuals and sets). RM provides several primitive commands that allows these tasks to be accomplished. The basic commands relevant to the design of functions are: define-set, define-variable, define-set-variable, define-binary-relation and define-function.

As an example consider the following RM command:

define-variable r1 rule

This first command defines a new variable of type **rule**. Since *rule* is a primitive types, RM, using frame-based property-inheritance, is able to determine what the properties of this newly defined object should be.

It is also possible to define variables over *user-defined* objects. Consider, for example, the following RM commands:

define-set Misdiagnosed-Cases {case| (/= (pdx case) (cdx case))}

define-set-variable mcases Misdiagnosed-Cases

define-variable mcase Misdiagnosed-Cases

The first command defines a new set **misdiagnosed-cases**, i.e., the set of all **case** such that **(Pdx case)≠(Cdx case)**, in English: the set of all **case** such that the known or expert conclusion is not identical to the knowledge base's most confident conclusion. The second and third command then establish **mcases** and **mcase** as variables over *subsets* of **misdiagnosed-cases**, and over the individual cases in **misdiagnosed-cases**, respectively.

Note that **misdiagnosed-cases** utilizes a function that is an implicit function of the active knowledge base, viz. **Cdx(case)**. Therefore the membership of the set **misdiagnosed-cases** will also vary as the active knowledge base changes. RM automatically sees to it that the membership of this set, as well as any other user-defined set whose membership is sensitive to the active knowledge base, is recomputed when a new knowledge base is activated.

As an illustration of a RM's function-definition capability, consider the following useful definition:

[define-function satisfied-rules-for-hypothesis (hypothesis case)
{rule in (rules-for hypothesis)| (= (satisfied rule case) 1)}].

This command defines a function **satisfied-rules-for-hypothesis** of two arguments, of type **hypothesis** and **case** respectively, which returns the set of all rules with conclusion **hypothesis** that are satisfied in **case**.

4. Defining Useful Refinement Concepts

We are now in position to see how useful refinement concepts may be defined using RM. As an example we choose a function used in SEEK2 called GenCF(rule), which defines a pattern of behavior a rule must exhibit in order to be considered for a possible "confidence-boosting" refinement. **GenCF(rule)** is the number of **mcase** in which, a) **rule** is satisfied and concludes the *correct conclusion* for **mcase**, i.e., **PDX(mcase)**, and b) of all the rules meeting clause (a), **rule** has the greatest confidence factor. (Intuitively, we are interested in the satisfied rule whose confidence will have to be raised the *least* in order to correct the mcase).

This concept is rendered in RM in two steps as follows:

```
[define-function GenCF-rule (mcase)
  SELECT rule IN
        (satisfied-rules-for-hypothesis (pdx mcase) mcase)
    WITH-MAX (rule-cf rule)]

(In English: given mcase, select a rule among
the satisfied rules for the correct conclusion
of mcase that has maximum confidence-factor.)

define-function GenCF (rule)
      |{mcase| (= rule (GenCF-rule mcase))}|

(In English: the cardinality of the set
of mcase in which rule is returned by
GenCF-rule(mcase).)
```

Continuing with this example allows us to exhibit something of the flexibility of the metalinguistic approach, and to show its use as both a customization device and a tool for experimental research. In general there may be a number of rules that satisfy the conditions given in **GenCF-rule(mcase)**. The motivation for SEEK2's selecting only one of these as *the* "genCF-rule" of **mcase**, has to do with our expectations concerning the size of the domain knowledge base, and our desire to keep the number of refinement experiments attempted relatively small [3]. Under other circumstances, however, it may be reasonable to "credit" *every* rule that satisfies these conditions as a "genCF-rule," and thus allow for the generation of a larger number of "confidence boosting" refinement experiments. In RM this is easily accomplished by altering the first definition given above in the following manner:

```
[define-function GenCF-rules (mcase)
  SELECT {rule} IN
        (satisfied-rules-for-hypothesis (pdx mcase) mcase)
    WITH-MAX (rule-cf rule)]

(In English: given mcase, select the set of rules
among the satisfied rules for the correct
conclusion of mcase that have the maximum
confidence-factor.)
```

5. Primitives for Supporting Heuristic Refinement Generation

According to the general model of heuristic refinement generation [7, 2], specific plausible rule refinements are generated by evaluating heuristics that relate the observed behavior and structural properties of rules to appropriate classes of rule refinements. Such heuristics will utilize refinement concepts or functions of the sort described above and elsewhere [3]. Here is an example of how such a heuristic is defined in RM:

```
[define-heuristic
  (if (> (GenCF rule) 0)
            (operation raise-cf rule
                  (mean-cdx-cf (GenCf-mcases rule))))]

(English translation: if Gen-Cf(rule) > 0 then
raise the confidence factor of rule to the
mean-value of CDX(mcase) over the mcases
that contribute to GenCF(rule), i.e., try
to "win" some of the mcases contributing to
GenCf(rule) by boosting the confidence-factor
of rule to the mean-value of the currently
incorrect conclusion in these mcases.)
```

The above heuristic is actually a simplified version of a similar heuristic currently in use in SEEK2. **mean-cdx-cf(cases)** and **GenCF-mcases(rule)** are also functions defined in terms of RM primitives.

6. Defining Control Strategies for Experimentation and Selection

A fully automatic refinement system will not only *generate* refinement suggestions, it will also test them over the current data base of cases, and, perhaps, create refined versions of the given knowledge base, $\mathbf{kb_0}$, that incorporates one or more tested refinements. A system that tentatively alters $\mathbf{kb_0}$ in an attempt to open up new refinement possibilities will be called a *generational* refinement system.

SEEK2 is an example of what we may call a *single-generation* refinement system: it is a generational system that keeps only one refined version of $\mathbf{kb_0}$ at any given time. A refinement system will be said to be a *multiple-generation* system if it is *capable* of creating and accessing several versions of the knowledge base at any given time.

In order to support the specification of such fully automated refinement systems, a refinement metalanguage must have primitives for evaluating heuristics, trying suggested refinement experiments, and creating new versions of the knowledge base if desired. The metalanguage should also allow the user to put all these primitive actions together into an overall control strategy for the refinement process.

There are several salient RM primitives for performing these tasks. One of these is a procedure, **Evaluate-heuristics(rules)**, that evaluates all the refinement heuristics for each rule in **rules**. This procedure may also take an optional argument specifying a particular subset of the heuristics to be evaluated as opposed to the entire set.

Suppose that $\mathbf{kb_a}$ is the *active knowledge base* (see section IV.A above). **Try-Experiment(operation)** is a procedure that 1) applies the refinement, **operation**, to $\mathbf{kb_a}$, 2) calculates the result of running this refined version of $\mathbf{kb_a}$ over all the cases, and returns a data-structure, called a *case-vector*, containing the new results, and 3) applies the *inverse* of **operation** to $\mathbf{kb_a}$ in order to return it to its original form. Thus this procedure does not change $\mathbf{kb_a}$ permanently; rather the case-vector it returns is used to determine the effectiveness of **operation**. By comparing this returned case-vector with the case-vector for $\mathbf{kb_a}$ one can determine the exact effect of **operation** on a case-by-case basis, if desired.

In order to create a refined knowledge base that can be available for future analysis the procedure **Create-Kb(operation)** must be invoked. This procedure "creates a knowledge base" **kb** that is the result of applying **operation** to $\mathbf{kb_a}$. The data structure that represents **kb** contains a) **operation**, b) a pointer to $\mathbf{kb_a}$, c) a slot for pointers to **kb**'s potential successors, d) a table summarizing the basic performance characteristics of this knowledge base, including, for example, the total number of cases this knowledge base diagnoses correctly.

In virtue of these predecessors and successors links, at any time the set of available knowledge bases forms a tree rooted at $\mathbf{kb_0}$. When RM is instructed to *activate* a knowledge base $\mathbf{kb} \neq \mathbf{kb_a}$, RM traces back through the ancestors of **kb** until either $\mathbf{kb_a}$ or $\mathbf{kb_0}$ is reached (one of these events must occur). If $\mathbf{kb_a}$ is reached then in order to activate **kb** all the refinement-operations occurring in the path from $\mathbf{kb_a}$ to **kb** are performed on the current internal version of the knowledge base. If $\mathbf{kb_0}$ is reached, the operator information in the path from $\mathbf{kb_0}$ to **kb** is used to activate **kb**. (Note that RM never requires more than one internal copy of the knowledge base; to activate a refined version of $\mathbf{kb_a}$, the current internal copy is modified in the specified manner using the information in the "tree of knowledge bases.")

To see how these primitives may be used to specify alternative control strategies, let us first briefly review SEEK2's control strategy. SEEK2 employs a cyclic control strategy that is a form of hill-climbing. In each cycle of operation the system a) evaluates its heuristics to generate refinement experiments, b) attempts all of these experiments, keeping track of the one that yields the greatest net gain in overall performance over the data base of cases, c) creates and activates a version of the knowledge base that incorporates the refinement yielding the greatest net gain. These cycles continue until a point is reached at which none of the generated refinements yield a postive net gain.

There are myriad ways in which this simple procedure can be modified [2]. As a simple example, suppose in a given cycle the system finds $n > 1$ refinements all yielding the same maximum net gain. One might wish to create n new knowledge bases, one for each of these refinements, and continue the process for each of these. Or perhaps one would like to incorporate a subset of these n refinements in $\mathbf{kb_a}$, rather than just one of them**. Using the primitive procedures we have discussed, such variations on the simple hill-climbing approach are easily specified in RM.

7. Incorporation of Domain-Specific Metaknowledge

In order to see how RM can be used to express domain-specific metaknowledge, we show how RM can be used to incorporate some of the important features of the approach discussed in [8].

Some rules in a knowledge base (or their supporting beliefs) may be *definitional* in character, or may represent laws or principles of a *theoretical* nature. As discussed in [8], it is reasonable to avoid modifications of such rules or beliefs, provided other refinement candidates can be identified. To incorporate this preference in one's refinement system using RM, one would first define the relevant sets using enumeration. For example,

define-set Definitional-Rules {4 5 15 27}

define-set Theoretical-Rules {1 2 11}

establishes rules 4, 5, 15, and, 27 as belonging to a set called "Definitional-rules," and rules 1, 2, and 11 as belonging to a set called "Theoretical-rules." One then has several options. One could cause the refinement system to avoid gathering information for the rules in these sets altogether by modifying the definitions of refinement concepts such as GenCF-rule (see section IV.B.4), e.g., GenCF-rule(mcase) may be defined as the *non-definitional* and *non-theoretical* rule such that etc. Or one could design the refinement system's control strategy so that experiments generated for rules belonging in one of these sets would be attempted only if no experiments to other rules are found in a given refinement cycle or session.

Another important idea contained in the approach to rule refinement discussed in [8], is the notion of a set of rules or beliefs *justifying* another rule or belief. While [8] presents a variety of attributes of "the justification relation" that are of potential use in knowledge base refinement, here we will only show how the basic idea can be incorporated in refinement systems specifiable using RM. Consider the following RM commands:

define-binary-relation Justifies (rules rule)

Assert Justifies {25 96} 101

The first command informs RM that the user intends to supply and make use of ordered-pairs of the form **<rules,rule>** under the relation-name "Justifies," i.e., ordered-pairs whose first element is a set of rules and whose second element is a rule. The second command then makes the assertion that the ordered-pair <{25 96},101> belongs to this relation. Intuitively, this represents the fact that rules 25 and 96 together provide a justification for rule 101.

V Some Concrete Results

A version of SEEK2 has been specified in RM. The RM definitions and commands for doing so take up roughly 4 pages of readable text. By contrast, the hard-coded implementation of SEEK2 takes up about 50 pages of code. This is evidence that the primitives in RM have been well-chosen: using them it is possible to write compact, easily understandable, but, nevertheless, powerful, specifications. As we claimed earlier, a well-designed metalinguistic system allows one to easily specify *what* one wants done, without having to worry about the details of *how* it is achieved.

In [3] we reported that the hard-coded version of SEEK2, working on a rheumatology knowledge base of 140 rules and 121 cases, was able to improve performance from a level of 73% (88 out of 121 cases diagnosed correctly) to a level of 98% (119 out of 121 cases diagnosed correctly) in roughly 18 minutes of CPU time on a VAX-785. The RM version of SEEK2 achieves the same results on the same machine in roughly 2 hours of CPU time. This result is much better than was actually anticipated since no attempt has been made to optimize the lisp-code translations RM generates from its high-level specifications (beyond any optimizations that may be attributable to the common-lisp compiler).

In addition, RM has been used to formulate and test alternative control strategies. For example, a modified version of SEEK2's control regime has been tested using the aforementioned knowledge base. Briefly, instead of selecting only the single refinement yielding the greatest net gain in a cycle, the modified control strategy allows for the selection of many successful refinements within a single cycle. For the knowledge base in question, this procedure converges to the same result as the simple hill-climbing approach, but does so in 1 hour of CPU time.

RM has also proven itself to be useful as both a debugging and design tool. Logical bugs in SEEK2 have been discovered through the use of RM, and a number of new refinement concepts and heuristics have recently been added to SEEK2 as a result of experimentation with RM [2].

RM has also been used as a tool for performing experiments concerning the statistical validity of the empirically-based heuristic approach to refinement generation employed in SEEK2***.

**It should be noted that there is no logical guarantee that incorporation of several *independently tested* refinements will yield a positive net gain equal to the sum of their individual gains, or even yield a gain at all [2]. Therefore, implementation of this tactic behooves one to retest the refinements again *in tandem* to be certain of a beneficial effect.

***The results of these experiments are encouraging and are reported in [2].

Following an accepted practice in statistical pattern recognition [1], to test a refinement system one can partition the set of domain cases into disjoint *training* and *testing* sets, and run the refinement system only on the training set. Afterwards, the resulting refined knowledge base(s) are run on the testing set. Such experiments are easily specified using RM.

VI Summary

A metalinguistic approach to the construction of knowledge base refinement systems has been described, motivated, and implemented. Concrete positive results have been achieved using a system based on this approach. In terms of future directions, it is reasonable to expect that similar metalinguistic approaches will be useful in designing more powerful refinement systems - e.g., systems that include a rule *acquisition* capability [4]- and that many of the key features of the metalanguage described here will be applicable to the design of metalanguages corresponding to richer object-languages.

The research on a metalinguistic framework presented here may be seen as an exploration of the consequences of applying the "knowledge is power" principle to the domain of knowledge acquistion itself, and more specifically, to knowledge base refinement. If domain knowledge gives a system problem-solving power, and if the domain of interest is itself the problem of making a given knowledge base fit certain given facts more closely, then it follows that *metaknowledge about knowledge representation itself* - e.g., knowledge of the ways in which formal objects can be used or altered to fit facts, knowledge of the sorts of evidence that can be gathered in support of certain classes of refinements, etc. - must be an essential ingredient of any successful automatic knowledge base refinement system. It also follows that just as there is a knowledge acquisition problem for ordinary "object-level" systems, so too there must be a *metaknowledge acquisition* problem for knowledge refinement and acquisition systems. Therefore, just as the use of high-level formal languages has helped researchers to clarify issues and generalize from experiences with object-level knowledge acquisition, one would expect that the use of a high-level *metalanguage* would provide similar benefits with respect to the metaknowledge acquisition problem. It is hoped that the work presented here will be seen as justifying this expectation.

VII Acknowledgments

I want to thank Sholom Weiss, Casimir Kulikowski, Peter Politakis, and Alex Borgida for helpful criticisms of this work.

References

1. Fukunaga, K.. *Introduction to Statistical Pattern Recognition.* Academic Press, New York, 1972.

2. Ginsberg, A. *Refinement of Expert System Knowledge Bases: A Metalinguistic Framework for Heuristic Analysis.* Ph.D. Th., Department of Computer Science, Rutgers University, 1986.

3. Ginsberg,A., Weiss,S., and Politakis,P. SEEK2: A Generalized Approach to Automatic Knowledge Base Refinement. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, 1985, pp. 367-374.

4. Kahn, G., Nowlan, S., Mcdermott, J. MORE: An Intelligent Knowledge Acquisition Tool. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, CA, 1985, pp. 581-584.

5. Lenat, D. The Role of Heuristics in Learning By Discovery: Three Case Studies. In *Machine Learning*, Tioga Publishing Company, 1983.

6. Mitchell, T. "Generalization as Search". *Artificial Intelligence 18* (1982), 203-226.

7. Politakis, P. and Weiss, S. "Using Empirical Analysis to Refine Expert System Knowledge Bases". *Artificial Intelligence 22* (1984), 23-48.

8. Smith, R., Winston, H., Mitchell, T., and Buchanan, B. Representation and Use of Explicit Justification for Knowledge Base Refinement. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, Los Angeles, California, 1985, pp. 673-680.

9. Weiss, S., and Kulikowski, C. EXPERT: A System for Developing Consultation Models. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, Tokyo, Japan, 1979, pp. 942-947.

10. Weiss, S., Kulikowski, C., Amarel, S., and Safir, A. "A Model-Based Method for Computer-aided Medical Decision-Making". *Artificial Intelligence 11*, 1,2 (August 1978), 145-172.