BOSWELL, R.A. 1998 *Knowledge refinement for a formulation system*. Robert Gordon University, PhD thesis. Hosted on OpenAIR [online]. Available from: https://doi.org/10.48526/rgu-wt-2807305

Knowledge refinement for a formulation system.

BOSWELL, R.A.

1998

The author of this thesis retains the right to be identified as such on any occasion in which content from this thesis is referenced or re-used. The licence under which this thesis is distributed applies to the text and any original images only – re-use of any third-party content must still be cleared with the original copyright holder.

OpenAIR @RGU

This document was downloaded from https://openair.rgu.ac.uk

CC BY NC

Knowledge Refinement for a Formulation System

Robin Alexander Boswell

A thesis submitted in partial fulfilment of the
requirements of
The Robert Gordon University
for the degree of Doctor of Philosophy

February 1998

Collaborating establishment: Zeneca Pharmaceuticals, Macclesfield

BRITISH LIBRARY

IMAGING SERVICES NORTH

Boston Spa, Wetherby
West Yorkshire, LS23 7BQ
www.bl.uk

BEST COPY AVAILABLE.

VARIABLE PRINT QUALITY

Abstract

This thesis describes the application of the knowledge refinement tool KRUST to the design system TFS, whose task is tablet formulation for a major pharmaceutical company. KRUST has already been successfully applied to a variety of classificatory problems, and a generic refinement framework is being developed. This thesis explores the differences in knowledge content and problem-solving steps for design rather than diagnosis systems, and how this affects the refinement process. It describes how novel components found in the design system were included within KRUST's underlying knowledge model, and how KRUST's refinement mechanisms were extended to apply to the design system by adding new operators to the existing tool-sets. Following this necessary adaptation of KRUST, new mechanisms were introduced whereby inductive learning from proofs of related examples is used to constrain and guide KRUST's refinement generation.

The concept of a generic refinement tool is introduced. In the course of the work described here, KRUST's knowledge and operator representations have developed in a way that facilitate its future application to different shells. The successful application of KRUST to TFS is used to show that KRUST has grown nearer to being a truly generic tool, and provides evidence that the construction of such a tool is both feasible and desirable.

Lastly, the role of knowledge refinement within software development is explored. Traditionally, refinement has been applied only to debugging, but the thesis shows how refinement can also play a role in software maintenance. In the course of its development, TFS has undergone both routine debugging, and also maintenance, when the formulation task was altered by a change in company policy. It was thus possible to test the extent to which KRUST was able to reproduce automatically the changes that were originally made manually to TFS, and hence to evaluate KRUST's effectiveness in both debugging and maintenance roles.

BRN: 20552			
AC		KE	
AT		ME	
CL	✓	WM	
FH			
HI			

Acknowledgments

I wish to thank my supervisor, Dr. Susan Craw, for her advice, guidance and patience throughout this project. She has been helpful in allowing me time off from the KRUSTWorks project to complete the work, and she has also been sufficiently trusting to permit me the use of her office and computer for uninterrupted study out of hours. Moreover, she and her husband have extended their hospitality to me on numerous occasions,

I am also grateful to Gareth Palmer for his patience with my frequent questions about the obscurer aspects of KRUST during my initial efforts at modifying it.

I thank Zeneca Pharmaceuticals for the use of the TFS software, and for the opportunity to visit their site in order to run a commercially-sensitive version. I am particularly grateful to Dr Ray Rowe for his enthusiastic support for the project. He has been most helpful in explaining the formulation process in terms comprehensible to a non-chemist, and his evaluation of KRUST's results from a pharmacologist's viewpoint were essential to the project's success.

I thank Logica PLC for the use of the PFES shell, and Paul Bentley for his assistance with PFES, and for making necessary modifications to the interface.

Finally, I thank the choir and congregation of St Margaret's church, Gallowgate, for offering a less computer-centered view of the world, and for the preservation of my sanity during the past three years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the background, motivation and objectives of the work described in this thesis. First, it explains the nature and purpose of knowledge base refinement (section 1.1). It then points out weaknesses in existing refinement tools (section 1.2), and outlines a program of work for developing a tool which overcomes some of these weaknesses (section 1.3). It shows that the refinement tool KRUST is an appropriate starting point for the development of such a tool, and it then introduces an industrial application, the tablet formulation system, which will be used to evaluate the tool. It next outlines the main work of the thesis: the application of KRUST to the tablet formulation system, and the implementation of inductive operators within KRUST.

There follows a discussion of the different ways in which refinement can assist in software development, and methods for the evaluation of the effectiveness of a refinement tool. The chapter then concludes with a synopsis of the thesis (section 1.4).

1.1 Knowledge-base refinement

When a knowledge base (KB) is first elicited, by whatever combination of interviews, knowledge acquisition tools and induction, it is unlikely to perform to the standard of a domain expert — hence the need for refinement. In the absence of an automatic tool, refinement traditionally requires the intensive use of both domain expert and knowledge engineer. Hence any tool which reduces the human input to this task will be useful; even an interactive, only partially automatic tool can offer a significant saving in human resources.

There are two approaches to the refinement of a knowledge-based system (KBS).

- **Static analysis** scans the KB for missing or inconsistent knowledge.
- **Dynamic analysis** runs the KBS on examples to identify faulty behaviour and then suggest suitable repairs.

The work described in this thesis falls under the heading of dynamic analysis. Formally, the process of dynamic refinement of KBSs may be expressed as follows. It requires a KBS containing a set of rules R, and a set of examples E for which the correct system behaviour is known. It may be assumed that the KBS performs incorrectly for some of the examples, or else no refinement is necessary. The goal of knowledge refinement is to construct a modified rule-base R' which performs more accurately on E than R does (ideally with 100% accuracy, but in practice this is rarely attainable). It is also usual to make the assumption that R is not grossly inaccurate, so that the changes to be made to it are relatively small, or "conservative"; and given a choice between refinements, the more conservative is usually preferred. The justification for the desire for conservatism is the requirement that the rules in the refined KB should be comprehensible and acceptable to the expert who helped to create the original KB.

Typically, knowledge refinement has been applied to classificatory systems, and the only significant aspects of "system behaviour" are the classes to which the KBS assigns examples. However, at least one refinement system allows the user to specify the order in which the KBS should perform certain actions; if the actions are performed in the wrong order, this is regarded as incorrect system behaviour (Murphy & Pazzani 1994). Furthermore, if a KBS generates more complex output than a simple class value, any deviation from the desired output will be regarded as incorrect behaviour.

Because of the complexity of the refinement task, there is currently no refinement tool which claims always to find the best possible refinement; that is, to generate a refined rule-base which will produce the most correct behaviour for the entire example set. Rather, all tools employ some sort of heuristic search which in practice is found to generate useful refinements, but is not guaranteed to find the optimal one. This explains the existence of a wide variety of different refinement tools, each using slightly different operators and search techniques.

1.2 Weaknesses in existing refinement tools

Existing knowledge refinement tools have a number of limitations.

- The KBSs to which they have been applied are mostly artificial, or "toy", and the faults in these rule bases have been artificially introduced, rather than occurring naturally in the course of development.
- The KBSs to which they have been applied solve classificatory problems, e.g., fault diagnosis. A number of KBSs solve a significantly different type of problem, that of design, but I am not aware of any application of a refinement tool to a design system, apart from the work described in this thesis.
- The tools refine backward-chaining rules only, with the exception of Clips-R (Murphy & Pazzani 1994).
- Each tool is applicable only to a single expert system shell.

These weaknesses are to some extent related. The origins of knowledge refinement in the debugging of PROLOG programs may be responsible for the fact that almost all refinement systems are applicable only to backward-chaining rules. This in turn means that the KBSs being refined will be mostly classificatory, since expert systems which perform design tasks are more naturally written using forward-chaining rules.

1.3 Objectives of the thesis

The objectives are derived from the weaknesses just identified in existing refinement tools. The principal objective is the development of a more practical refinement tool which could be applied to an expert system developed and used in industry, and which could assist with software development by identifying and suggesting fixes for faults. Secondary objectives are:

- to demonstrate that the tool is able to refine a design system, as well as classificatory ones, and
- to create a generic, extensible refinement tool, capable of refining a variety of different shells.

Given these goals, some decisions had to be made. First of all, a refinement tool could be created from scratch, or an existing tool adapted. Since the objectives were to *extend* the power of current tools in various ways, it would clearly be a more efficient use of time to start with an existing tool. The KRUST refinement tool (Craw 1991) proved a suitable starting point for the proposed work, for the reasons given below.

A useful distinction between types of refinement systems is made by Craw, Sleeman, Boswell & Carbonara (1994), who distinguish Knowledge Base Refinement (KBR) from Theory Revision (TR). For the purposes of the present discussion, the most important features separating the two are

Domains: KBR refines potentially noisy expert systems, whereas TR refines KBS which are more complex, possibly involving recursion, but which are noise-free.

Control Strategies: KBR refines expert systems using a variety of control and conflict resolution strategies, whereas TR is usually applied only to PROLOG programs.

Testing: KBR tools typically require fewer examples than TR tools.

It follows that a KBR tool would be more appropriate to my goals than a TR tool, and KRUST's properties place it clearly among the KBR tools. Moreover, KRUST has several other features, such as its knowledge and operator representations, which fitted it for application to industrial systems, and for development into a generic refinement tool. Moreover, KRUST has been found to be competitive with other systems when applied to a number of artificial problems. Consequently, I concluded that my goals could best be achieved by building on previous KRUST projects, further developing KRUST in such a way that it could be applied to an industrial expert system.

The next step was obtaining an expert system to apply KRUST to, ideally a design system. I was fortunate in obtaining just such a system from Zeneca Pharmaceuticals: the Tablet Formulation System (TFS). In addition, the software firm Logica PLC provided the Product Formulation Expert System Shell (PFES); this is the language and environment in which TFS was written. PFES is a forward-chaining, task-based shell designed for solving design and formulation problems. The domain of TFS, tablet formulation, is the task of choosing inert substances, or excipients, which need to be mixed with a drug in order to create a tablet having the required physical properties. The task is one of constructing a

formulation which simultaneously satisfies a number of constraints, often conflicting, and is therefore a design problem. TFS has been in regular use by formulators at Zeneca for some years, so satisfies the requirement for a "real" expert system.

1.3.1 The adaptation of KRUST to the tablet formulation system

Given the choice of KRUST as a starting point, and the tablet formulation system as an application, the manner in which the objectives of the thesis may be achieved can be stated more precisely; KRUST must be extended in a generic fashion so that it can be used to refine TFS.

The goal of developing a generic tool could be at least partially fulfilled by developing a consistent and extensible framework for knowledge and operator representation. The generic nature of the resulting tool could then be proved by its effectiveness on at least two environments: the shell for which KRUST was originally designed (PROLOG) and the new shell, PFES. Moreover, in the course of the work described in this thesis, KRUST was extended by others to apply to the CLIPS and POWERMODEL shells, thus further justifying my claim that my work fitted in with the general goal of developing a generic tool.

1.3.2 The implementation of inductive operators for KRUST

The review of related work revealed a weakness in knowledge base refinement systems, and KRUST in particular: that they are less able than theory revision systems to make use of induction. KRUST in particular had no inductive operators at all. This reduced the rules and conditions that KRUST was able to learn, and hence the faults that it was able to fix. Therefore, the decision to base my work on KRUST imposed an additional goal if the resulting refinement system were to be competitive: that of adding inductive operators to KRUST.

KRUST's weakness in the area of induction was overcome by the addition of a number of refinement operators which were able to learn from multiple examples, and to create rules and conditions which could not have been learned by the original KRUST.

1.3.3 The role of knowledge refinement in software development

The purpose of obtaining an industrial expert system was to evaluate the effectiveness of my work on KRUST; I hoped to show that KRUST could be extended to apply to a new shell, in this case PFES, and that it would then be able to assist in software development by identifying and fixing faults. At the time when I first made contact with Zeneca Pharmaceuticals, several versions of TFS existed. One version, TFS-1B, simply fixed bugs in the previous versions. A later version, TFS-2, implemented a significant change in the specification of the formulation task. Development of TFS was then almost complete, and Zeneca would not allow access to the very latest version, TFS-3, so that KRUST could not be employed during the actual development of TFS. However, the existence of older versions provided a realistic solution. KRUST could be applied to the older versions, to determine whether it was able to fix the bugs that had actually occurred in the course of development.

Moreover, the experience with TFS lead to an expansion of my ideas on the role of knowledge refinement within software development. It is generally believed that the role of knowledge base refinement within software development is that of debugging. It became apparent during the work described here that refinement could also assist with software maintenance. Debugging is concerned with fixing faults where the behaviour of a KBS differs from its specification; maintenance is the harder task of modifying a KBS to match a new or changed specification. The degree of change to a specification for which the associated software modification is to be regarded as maintenance is arguable, and to some extent subjective. For the purposes of this thesis, a change in specification will be regarded as requiring maintenance, and not just debugging, if it requires a significant time, of the order of several man-months at least, to complete, and is regarded by a domain expert as a major change in the way the software carries out its task.

The successful application of knowledge refinement to software maintenance would constitute a significant advance.

1.3.4 Evaluation

The decision to emphasise the applicability of my work to industrial systems affected the choice of an appropriate method of evaluation. Evaluation methods originating in

the machine learning community, such as repeated partitioning of an example set into training and testing examples, followed by cross-validation, become less useful. A more appropriate test is the ability to fix real faults which occur during the development of the system. When refinement is employed in a maintenance role, the test is its ability to modify the KBS to match the new specification.

The choice of a design system as a target for refinement introduces a further difficulty. Evaluation is harder for design systems than for classificatory systems, because there is typically no one best solution for a design problem. This makes it more difficult both for a refinement system to select the best refinement, and for the user to evaluate the effectiveness of the refinement tool. One solution lies in the involvement of a domain expert in evaluation.

1.4 Synopsis

Chapter 1 has summarised the aims of this thesis. Chapter 2 reviews related work in knowledge base refinement, and also includes some reference to machine learning. The refinement and learning systems described are mainly symbolic, since these are most relevant to my own work, but hybrid symbolic/neural-net systems are also described, for purposes of comparison.

Chapter 3 then introduces the tablet formulation system which will be used to evaluate my work on KRUST. This chapter discusses formulation and design problems in general, and distinguishes them from the classificatory problems to which refinement systems have more usually been applied. It explains why formulation problems are hard, and how KBSs might be of use in solving them. It then introduces the Product Formulation Expert System (PFES), a shell designed for writing formulation tools, and the particular problem of tablet formulation. Finally, it describes the Tablet Formulation System itself.

Chapter 4 provides a necessary introduction to my own work by describing the original KRUST program in some detail. Given my long-term aims of developing a generic refinement tool, able to fix faults in industrial expert systems, I explain why KRUST was an appropriate starting point for this work. The next two chapters contain the bulk of my own work. Chapter 5 describes how I modified KRUST to make it applicable to TFS. The principal areas of KRUST where modification was required were communication with

the KBS, knowledge representation, and refinement operators. I describe how I extended KRUST's capabilities in these areas in a generic way, so that the lessons learned will make it easier to apply KRUST to other shells in future. Chapter 6 describes how I further extended KRUST's abilities by adding inductive operators, thus overcoming a significant weakness which had been identified by the literature review.

Chapter 7 describes some clustering techniques, and how example clustering is useful at various points in KRUST's refinement procedures. Some of these techniques are used in the evaluation of KRUST, which is described in chapter 8. This chapter discusses the difficulties of evaluating refinements to a design system, and describes and justifies my experimental design. It then presents the results obtained when KRUST was applied to two versions of TFS: TFS-1A and TFS-1B. The first task is one of debugging, and the second task is one of maintenance, and therefore harder. Both qualitative results, such a description of the refinements generated, and statistical results concerning the accuracy of the refined KBs, are presented.

The remainder of the thesis is devoted to comparison with other work, and conclusions. Chapter 9 compares my work with the program EITHER, chosen as a representative symbolic refinement program, and CLIPS-R, which refines CLIPS programs, and tackles many of the same problems as my own work on a PFES application. Finally, chapter 10 presents conclusions, and suggestions for future work.

Chapter 2

Related work

2.1 Introduction

This chapter presents a summary of current work in knowledge refinement. First, the programs to be discussed are listed and classified. The principal division is into *symbolic* and *hybrid* systems, The word *symbolic* refers to systems which represent knowledge explicitly in the form of rules and facts, and distinguishes such systems from sub-symbolic ones that use neural nets or genetic algorithms. The word *symbolic* does not exclude the presence of numeric elements in rules and facts. *Hybrid* systems are those that combine symbolic and sub-symbolic techniques. The principal hybrid systems, and the only ones considered here, use artificial neural nets as their sub-symbolic technique.

These two classes are then considered separately. The symbolic algorithms are treated in more detail, since they are more relevant to my own work, which is purely symbolic. The chapter concludes with a summary of weaknesses in existing work, which provides motivation for my own work.

2.2 Systems discussed in this chapter

Figure 2.1 classifies the systems to be discussed. It shows that the majority are symbolic refinement systems. In addition to these, several machine learning systems are included. The reason for the inclusion of FOIL (Quinlan 1990) and FOCL (Pazzani & Kibler 1990) is that both include generalisation and specialisation operators which are comparable with those used by refinement systems, and moreover they both function as modules

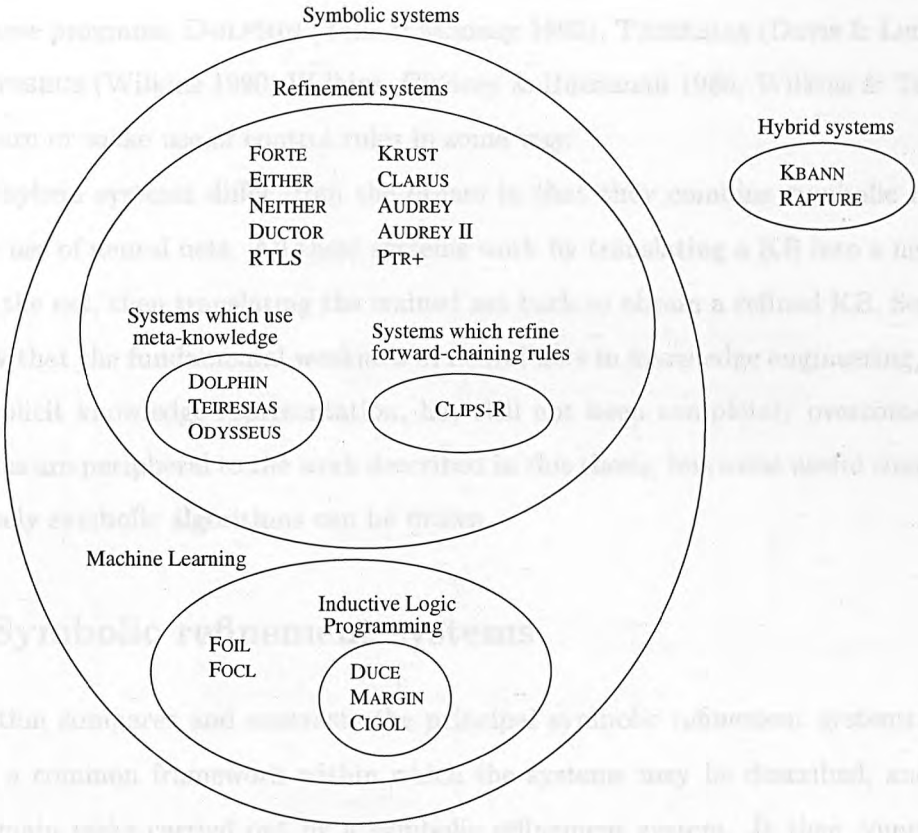

Figure 2.1: Classified algorithms

within a refinement system (RS). The inductive logic programming (ILP) systems are included because both they and many RSs include operators for inducing rules, but the ILP operators are significantly different from those of other systems.

The discussion of CLIPS-R (Murphy & Pazzani 1994) appears not in this section but in chapter 9. The reason is that CLIPS-R, which refines CLIPS KBSs, is the only RS apart from KRUST which is applicable to forward-chaining rules. Moreover, it includes two inductive operators. A complete account of CLIPS-R therefore needs to compare CLIPS-R's techniques with the techniques I have used to adapt KRUST to refine a PFES application, and with the inductive operators I have implemented. The account can therefore most usefully be presented *after* the description of the work done on KRUST.

Most of the programs shown in figure 2.1 learn or refine domain rules. It will become apparent that they are somewhat limited in their handling of control mechanisms or meta-rules, which suggests the desirability of overcoming that limitation in future work. The exceptions are the three programs described as "systems which use meta-knowledge".

These three programs, DOLPHIN (Zelle & Mooney 1993), TEIRESIAS (Davis & Lenat 1982) and ODYSSEUS (Wilkins 1990, Wilkins, Clancey & Buchanan 1986, Wilkins & Tan 1989), either learn or make use of control rules in some way.

The hybrid systems differ from the others in that they combine symbolic reasoning with the use of neural nets. All these systems work by translating a KB into a neural net, training the net, then translating the trained net back to obtain a refined KB. Section 2.5 will show that the fundamental weakness of neural nets in knowledge engineering, the lack of an explicit knowledge representation, has still not been completely overcome. These algorithms are peripheral to the work described in this thesis, but some useful comparisons with purely symbolic algorithms can be drawn.

2.3 Symbolic refinement systems

This section compares and contrasts the principal symbolic refinement systems. It first presents a common framework within which the systems may be described, and identifies the main tasks carried out by a symbolic refinement system. It then compares the approaches taken to each task by the different systems.

2.3.1 A generic refinement algorithm

Rule-based systems are liable to two sorts of errors: rules may fire when they should not, leading to false positives; or they may fail to fire when they should, leading to false negatives. At the highest level, all symbolic RSs iterate repeatedly through the following operations, in order to identify and fix both types of error (see Figure 2.2).

- **Blame Allocation** identifies which rules or conditions might be responsible for the erroneous behaviour of the KB.
- **Refinement Generation** generates one or more refinements for each of the potentially faulty rules or conditions, with the aim of fixing the erroneous behaviour.
- **Refinement Selection** chooses the best refinement(s).
- **Refinement Implementation** creates a new KB or KBs corresponding to the selected refinement(s).

Here a "refinement" is defined to be a combination of changes, where each change results from the application of a single operator, such as condition addition or deletion. The nature of refinements therefore varies from system to system, though in practice there is considerable overlap between the operators provided by each system (section 2.3.3).

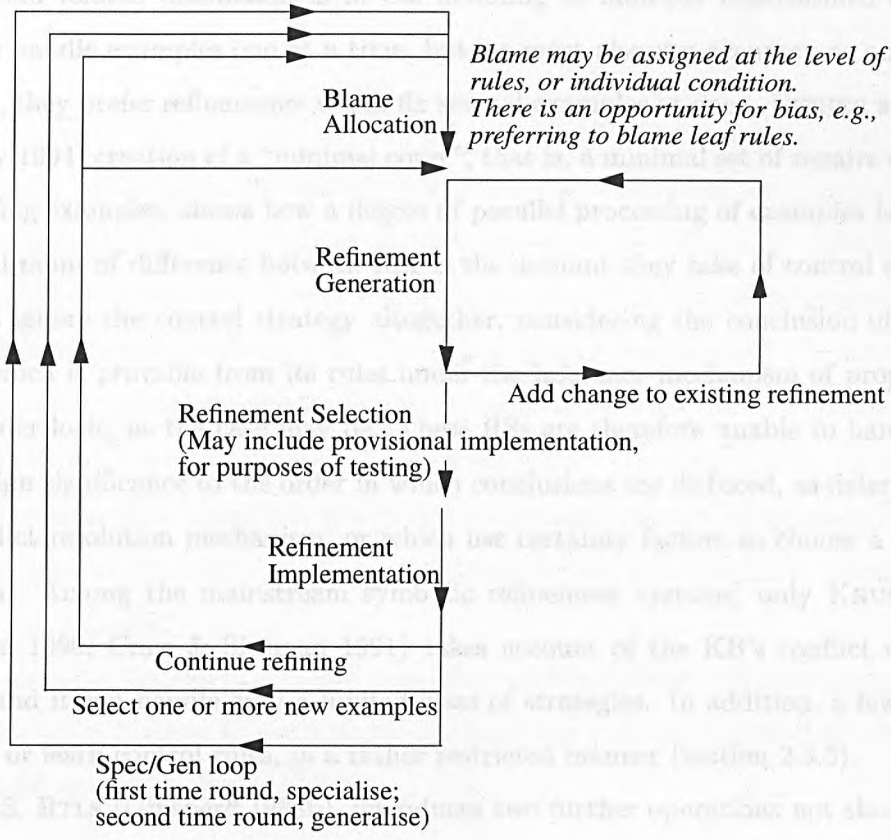

Figure 2.2: A generic symbolic refinement algorithm

Figure 2.2 has two main messages. First, the importance of the four operations: blame allocation, refinement generation, etc., listed down the centre of the figure. Secondly, that differences between algorithms can be expressed in terms of the order in which loops are nested. For example, systems vary in the complexity of the refinement generated at each pass down through the central series of arrows (refinement generation, selection and implementation). Some ensure before implementation that a refinement designed to fix a false negative does not introduce any new false positives as a side-effect; others permit such side-effects, and fix them in later iterations. This difference is reflected in Figure 2.2 in the relative proportion of iterations around the left-hand clockwise and right-hand

anti-clockwise loops.

Note that any particular algorithm need not make use of all the loops (indeed, none of the programs described does), and that the Spec/Gen loop could logically occur within the example selection loop, as well as outside it.

A second related distinction is in the handling of multiple misclassified examples. Most RSs handle examples one at a time, but, as most also use accuracy as one of their heuristics, they prefer refinements which fix several examples at once. EITHER's (Ourston & Mooney 1994) creation of a "minimal cover", that is, a minimal set of repairs which will fix all failing examples, shows how a degree of parallel processing of examples is possible.

A final point of difference between RSs is the account they take of control strategies. Most RSs ignore the control strategy altogether, considering the conclusion of a KB to be that which is provable from its rules under the inference mechanism of propositional or first order logic, as the case may be. These RSs are therefore unable to handle shells which assign significance to the order in which conclusions are deduced, as determined by their conflict resolution mechanism, or which use certainty factors to choose a preferred conclusion. Among the mainstream symbolic refinement systems, only KRUST (Craw & Sleeman 1990, Craw & Sleeman 1991) takes account of the KB's conflict resolution strategy, and it can handle only a limited class of strategies. In addition, a few systems either use or learn control rules, in a rather restricted manner (section 2.3.5).

One RS, RTLS (Ginsberg 1988b), introduces two further operations not shown in figure 2.2; these are pre and post-processes which need to be applied before and after refinement. Before RTLS is applied, the KB must be *reduced* or *flattened* (Ginsberg 1988b), and once RTLS has refined the flattened KB, it must then be retranslated into an unflattened form (Ginsberg 1990). *Flattening* here implies the removal of all intermediate results, so that a flattened KB consists of rules which express classes in terms of observables. Consider for example the following KB.

$$\tau_1 \ :- \ p.$$

$$\tau_2 \ :- \ q.$$

$$p \ :- \ a.$$

$$p \ :- \ b.$$

$$q \ :- \ c.$$

Suppose that a, b and c are observables, τ_1 and τ_2 are classes. Then the process of flattening creates the following rules.

$$\tau_1 \ :- \ a \vee b.$$

$$\tau_2 \ :- \ c.$$

The advantage of the initial flattening of the KB is that it reduces the complexity of the refinement process. The one drawback of this approach is the difficulty of retranslating the refined KB into an unflattened form. This is the problem that also poses the greatest difficulty for hybrid systems (section 2.5). There is no unique way of doing the re-translation, and the algorithm currently employed does not guarantee that the retranslated KB will behave in the same way as the KB output from RTLS. Ginsberg (1990) claims experiments have shown that in practice any changes introduced during re-translation result in an improvement in the performance of the KB, but he has been unable to prove that this will always be the case.

The section now considers in turn the various successive phases of knowledge refinement as shown in figure 2.2, and for each phase compares the different approaches taken by the refinement programs being considered.

2.3.2 Blame allocation

Figure 2.3 shows how the analysis of the proof tree and partial proof tree for a single example permits the assignment of blame to rules and conditions for both a false negative and a false positive. Each circle and rectangle represents a rule condition and/or conclusion. To the left of the figure is a proof of the incorrect conclusion H^S generated by the system; the grey circles represent conditions and conclusions which participated in this proof, and are therefore potentially to blame. To the right of the figure is a partial proof of the desired conclusion H^E; it is not a complete proof, because some of the conditions necessary for the proof of H^E are themselves not provable (represented as diamonds). These unprovable conditions are potentially to blame for the *failure* of the partial proof of H^E. Note that there is one currently unprovable condition which is shared between both proofs — this

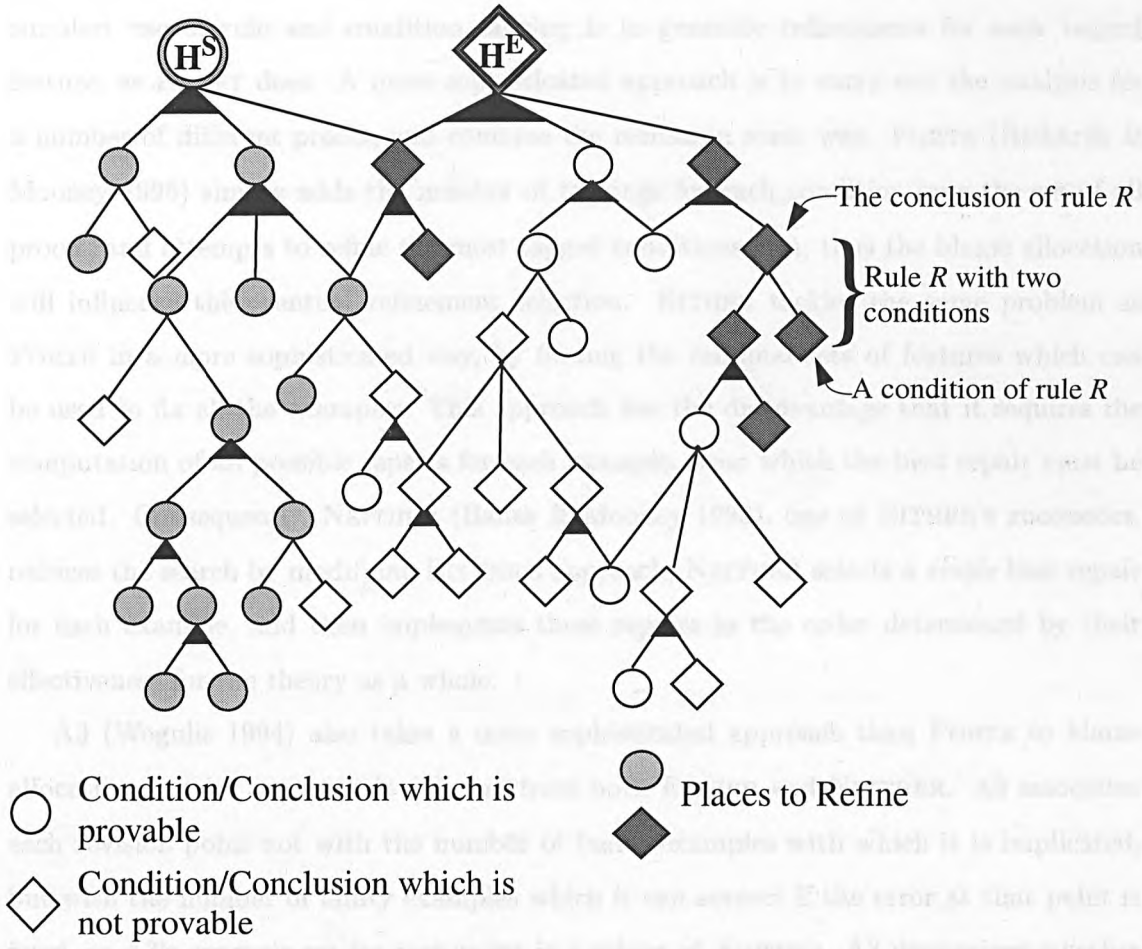

Figure 2.3: Solution graphs

illustrates the problem of potentially conflicting refinements, which will have to be dealt with later, at the refinement selection stage.

Figure 2.3 is incomplete as a depiction of the blame allocation process, in two ways:

1. it illustrates blame allocation for a single example only, and
2. for rules containing variables, blame needs to be spread further than is shown here.

Blame allocation for multiple examples

Figure 2.3 illustrates how it is possible to determine which rules and conditions may be responsible for the incorrect behaviour of a KBS. This process is commonly described as *tagging*. The grey conditions, and all the rules whose conclusions are grey, are potentially responsible for the incorrect conclusion of the proof-tree, so are said to be tagged. The

simplest use of rule and condition tagging is to generate refinements for each tagged feature, as KRUST does. A more sophisticated approach is to carry out the analysis for a number of different proofs, and combine the results in some way. FORTE (Richards & Mooney 1995) simply adds the number of taggings for each condition over the set of all proofs, and attempts to refine the most tagged conditions first; thus the blame allocation will influence the eventual refinement selection. EITHER tackles the same problem as FORTE in a more sophisticated way, by finding the minimal sets of features which can be used to fix all the examples. This approach has the disadvantage that it requires the computation of all possible repairs for each example, from which the best repair must be selected. Consequently, NEITHER (Baffes & Mooney 1993), one of EITHER's successors, reduces the search by modifying EITHER's approach; NEITHER selects a *single* best repair for each example, and then implements these repairs in the order determined by their effectiveness for the theory as a whole.

A3 (Wogulis 1994) also takes a more sophisticated approach than FORTE to blame allocation, but its approach is different from both EITHER and NEITHER. A3 associates each revision point not with the number of faulty examples with which it is implicated, but with the number of faulty examples which it can correct if the error at that point is fixed, so A3's example set for each point is a subset of FORTE's. A3 determines whether a change to the revision point can fix an associated example by making an *assumption* for the associated goal that is the opposite of the outcome supported by the theory, and then re-executing the KB. That is, if the goal succeeds, A3 re-executes the KB but causes the goal to fail, and *vice versa*. However, there is one danger in this approach; it will only identify errors that can be repaired by a single modification to the theory.

Finally, a comparison can be made with neural net algorithms. These in effect perform blame allocation and refinement generation simultaneously; more significantly, the symbolic algorithms use only faulty examples for blame allocation, whereas the network algorithms use positive examples to increase their belief in the correctness of a rule, as well as using negative examples to reduce it.

Blame allocation for first order rules

Given a KB written in first-order logic, a situation can arise where the range of potentially faulty conditions is more extensive than that shown in Figure 2.3. If a rule in a partial

proof of H^E fails to fire because one or more of its conditions fails, then the candidates for refinement include not only the failing conditions, but also any conditions which bind variables included in the failing conditions. For example, in Figure 2.4, predicate $p(X, Y)$ is under suspicion for the failure of $q(Y)$, even though the condition $p(X, Y)$ itself succeeded.

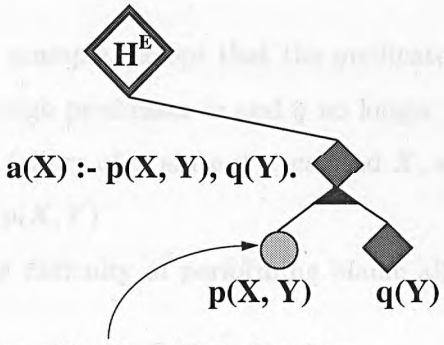

Figure 2.4: Blame allocation in first-order rules

It is possible to extend the example in Figure 2.4 to a situation in which a variable binding is responsible for a failure not in the same clause, but one or more clauses later. Consider for example rules 2.1 and 2.2.

$$v(X) \ :- \ w(X), a(X). \tag{2.1}$$

$$a(X) \ :- \ p(X, Y), q(X, Y). \tag{2.2}$$

Suppose the top-level goal is $v(X)$, with X unbound, and that $v(X)$ ought to succeed. Suppose however that $v(X)$ fails because $q(X, Y)$ fails, causing $a(X)$ to fail. Possible culprits for the incorrect behaviour should include not only the predicate p, which caused Y to be bound, but also w, which caused X to be bound.

Consider next a modified version of this example.

$$v(X) \ :- \ w(X), a(X).$$
$$a(X) \ :- \ p(X,Y), q(Y).$$

This is identical to the previous example, except that the predicate q does not take X as an argument. However, even though predicates w and q no longer share an argument, w could still be responsible for the failure of q, since it *does* bind X, and thus constrain the value bound to Y by the call to $p(X,Y)$

These examples illustrate the difficulty of performing blame allocation within proofs in first-order logic.

2.3.3 Refinement operators, and refinement generation

Once blame has been allocated as described in the previous section, a refinement system must attempt to modify the faulty rules, using a set of refinement operators. There are only a limited number of basic ways in which rules *can* be refined; any more complex refinement operators must be built from these primitives (the program NEITHER described below, is an exception). The primitive operators are:

Specialisation operators:

- Delete rule
- Add condition
- Specialise condition

Generalisation operators:

- Add rule
- Delete condition
- Generalise condition

Two points should be noted about these primitive operators.

- The capacity of a refinement system to specialise or generalise conditions is a consequence of the type of conditions used — currently only KRUST, EITHER and NEITHER manipulate rules whose conditions are liable to refinement. EITHER refines conditions of the form $k_1 < x < k_2$. KRUST refines both double-bounded inequalities such as these, as well as single bounded inequalities such as $x < k$ and $x > k$. KRUST also refines conditions which test the value of hierarchically-structured attributes. (This is described in more detail in chapter 4. See figure 4.1 for an example).

- NEITHER can refine what are known as *m-of-n* conditions. An *m-of-n* condition is a set of n conditions; the m-of-n condition is defined to be true if and only if at least m of the component conditions are true. The ability to refine m-of-n conditions enables NEITHER to make use of a refinement operator which is specific to that type of condition; this operator raises or lowers the threshold m and thus specialises or generalises the condition. This enables NEITHER to perform well in domains such as protein-folding which are well described by m-of-n conditions, doing better than systems like KRUST which do not make use of this representation. However, the presence of this representation and its associated operators is independent of other tasks within the refinement process, so that it would be relatively simple to add m-of-n rules to any of the other refinement systems. Until competing systems are provided with a "level playing-field" in terms of their knowledge representation and associated operators, it is impossible to assess to what extent their relative success is due to their underlying algorithms.

- The learning algorithms FOIL (Quinlan 1990) and FOCL start with an empty rule set, and so employ only the "add rule" and "add condition" operators.

All the algorithms except KRUST use repeated applications of these operators to perform more complex refinements. Most use a hill-climbing approach, which brings with it the risk of getting stuck on local maxima. FORTE is the one exception; if deleting conditions one at a time fails to produce a correct rule, it will then attempt to delete several conditions simultaneously. However, it adopts a different approach when *adding* conditions. If adding conditions singly fails to produce a correct rule, then rather than trying to add multiple conditions simultaneously, which would lead to a combinatorial explosion, it instead adopts a technique unique to FORTE called *relational pathfinding*. This is a

somewhat specialised technique applicable in domains where the initial theory contains a number of primitive relations between constants from which it is assumed that the target relation(s) may be constructed. Consider for example a "human relationship" domain containing a number of facts of the form *parent(X, Y)*, *brother(X, Y)*. Suppose FORTE is required to learn the concept of *uncle(X,Y)* from examples of the form $uncle(u_i, n_i)$. The method of relational pathfinding starts from each n_i and tries to find a sequence of relationships which will lead to the associated u_i. If the relationships are expressed as a graph, then pathfinding amounts to seeking a path through the graph from n_i to u_i. When such a path has been found, the primitive relationships making up the path may be combined to form a rule defining the new relationship, in this case, *uncle(X,Y)*. For example, figure 2.5 shows the path linking nephew s to uncle b. Once FORTE has found this path, it will use the two relationships making up the path to create the rule

$$uncle(X,Y) :-$$
$$brother(X,P),$$
$$parent(P,Y).$$

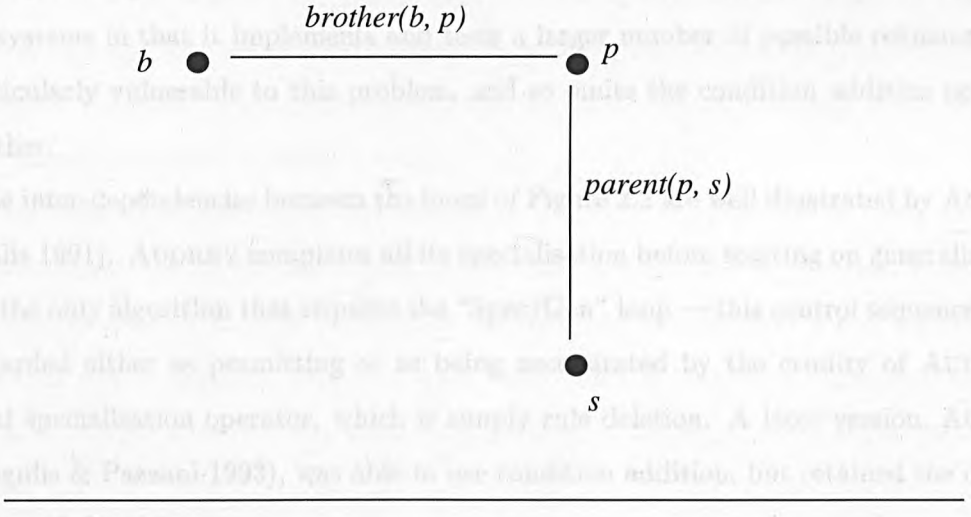

Figure 2.5: Relational data expressed as a graph

At a higher level, any initial over-generalisation in FORTE is detected and fixed immediately by combining the generalisation with further specialisation operators, and *vice versa*. The latter case is handled by generating a set of rules, each more specialised than

the initial over-general rule, and replacing the initial rule with the more specialised ones.

Rule addition

Although FORTE claims to have a rule-addition operator, this is really a rule-modification operator; FORTE takes a copy of an existing rule and then applies condition deletion and addition. DUCTOR (Cain 1991) on the other hand has a true rule addition operator. It obtains the conditions for a new rule by constructing a set of attributes that discriminates false negatives from true negatives, and removing those attributes that are used elsewhere in the proofs. EITHER uses a similar approach, but uses ID3 3 to perform the induction, and is thus better able to handle noise than DUCTOR, which simply identifies a discriminant set of attributes. The point to be noted here is that any algorithm lacking an inductive component is at a disadvantage when needing to apply an "add condition", and still more, an "add rule" operator. For example, KRUST totally lacks the "add condition" operator, and its "add rule" operator is crude; the conditions of the new rule are simply the conjunction of all the properties of the refinement example which caused the rule to be learned. Induction is not an *essential* component in condition addition. However, condition addition without induction is liable to lead to a combinatorial explosion because of the large number of possible new conditions to be considered. Since KRUST differs from most other systems in that it implements and tests a larger number of possible refinements, it is particularly vulnerable to this problem, and so omits the condition addition operator altogether.

The inter-dependencies between the loops of Figure 2.2 are well illustrated by AUDREY (Wogulis 1991). AUDREY completes all its specialisation before starting on generalisation, and is the only algorithm that requires the "Spec/Gen" loop — this control sequence could be regarded either as permitting or as being necessitated by the crudity of AUDREY's original specialisation operator, which is simply rule deletion. A later version, AUDREY II (Wogulis & Pazzani 1993), was able to use condition addition, but retained the control structure of AUDREY.

2.3.4 Refinement selection

Refinement selection is to some extent dependent on the preceding two operations. For example, the earlier versions of EITHER preferred to refine leaf rules. Although this is a bias that appears to affect refinement generation, for efficiency it is actually implemented as soon as possible; that is, at the blame allocation stage. Likewise, DUCTOR orders the rules in the proof of a false positive in what the authors call "rule post-ordering" at the refinement stage, in order that refinements to leaf-rules should be generated and selected first.

Similarly, the choice of refinement operators reflects a bias towards a particular type of refinement. The types of refinement preferred by many systems indicate an apparent conflict between the desire for radical refinements at the refinement generation stage as opposed to the desire for conservative refinements at the initial blame allocation stage. For example, several systems choose rule deletion as a specialisation operator before condition addition; and NEITHER prefers to generalise m-of-n rules by reducing the threshold, rather than deleting conditions, on the grounds that threshold reduction is "more aggressive", i.e., *less* conservative. A possible justification for this is that at the later refinement generation stage, unlike the blame allocation stage at which conservatism is preferred, it may be possible to evaluate the refinement in terms of misclassified examples fixed and new errors introduced; and a less conservative change may be justified if it scores well in these terms. After all, a truly conservative system would change nothing, so the policy of conservatism should not be taken too far.

In the case of systems that use sequences of primitive operators in the refinement generation stage, the separation between refinement generation and selection is less clear. For example, systems such as FORTE use hill-climbing to specialise a rule by adding a sequence of new conditions, and stop when they reach a plateau of whatever heuristic function is employed — thus the heuristic contributes to both refinement generation and selection.

Further heuristics may also be introduced at a later stage in the process such as accuracy on the training set, and/or on a further example set (a *pruning set*) noted as being particularly important, such as KRUST's "chestnuts" (section 4.7).

CLARUS (Brunk & Pazzani 1995) is unusual in using what the authors call "lexical

cohesiveness" as well as accuracy as a criterion for evaluating proposed new rules. The effect of this is to prefer rules where the meaning of the terms in the conditions and conclusion is closely related. For example, the rule

```
military_deferment(P) :-
    enlist(P, O),
    armed_forces(O).
```

would be preferred to

```
military_deferment(P) :-
    bankrupt(P),
    continuously_enrolled(P).
```

on the grounds that the meanings of the words "military", "deferment", "enlist", "armed" and "forces" are more closely related than "military", "bankrupt", etc. The "closeness" of the meanings of individual words is derived from the lexical database WORDNET (Beckwith, Fellbaum, Gross & Miller 1991).

2.3.5 Use of proofs, partial proofs, and control rules

All refinement systems make use of proofs and partial proofs at the blame allocation stage, as indicated in Figure 2.3. However, the following programs are of particular interest because of the way in which they make use of explicit control or meta-rules, either as input to the refinement process, or in its output. KRUST is included here because of its ability to take into account control information; this distinguishes it from the other symbolic refinement systems so far discussed. (A more detailed account of KRUST is presented in chapter 4).

KRUST refines backward-chaining rules, where conflict-resolution is performed by rule ordering. Thus KRUST must take account of this mechanism at the blame-allocation stage. In addition, KRUST is able to *use* this control mechanism at the refinement generation stage by changing the rule ordering, thus enabling rules to fire that previously did not, and *vice versa*.

DOLPHIN (Zelle & Mooney 1993) is described by the authors as learning control information. However, the information is expressed as extra conditions ("guards") which are added at the start of a rule, so in the terms of this survey, it may be regarded as a refinement system with a single refinement operator: condition addition. However, the purpose of the addition is not, as with the other systems, to achieve greater *accuracy*, but greater *efficiency* by preventing rule firings that do not lead to a solution. The extra condition does not alter the system's conclusion, it merely allows it to reach the conclusion more quickly. This is a modern illustration of the point made by Bundy, Silver & Plummer (1985) that what is "really" control information may be included in rules in a form syntactically indistinguishable from object-level conditions, so that tools which are only designed to refine object knowledge may end up refining control information as well.

TEIRESIAS and MYCIN (Davis & Lenat 1982). TEIRESIAS is a tool which assists in the interactive refinement of KBSs written using the shell MYCIN. Two forms of meta-rules are used by the MYCIN/TEIRESIAS combination; MYCIN makes use of its own meta-rules to assist in conflict resolution during the execution of its object rules, and TEIRESIAS makes use of *rule models* to critique new rules suggested by the domain expert. Rule models are induced automatically from the MYCIN rule-base, and recognise common patterns in the rules, such as the fact the attribute A always appears in conjunction with attribute B. Surprisingly, TEIRESIAS does *not* take any account of MYCIN's meta-rules when refining MYCIN rule-bases.

ODYSSEUS (Wilkins 1990, Wilkins et al. 1986, Wilkins & Tan 1989) is very much one of a kind, and illustrates a different approach to the use of control information from that of the other programs surveyed. ODYSSEUS is an apprentice learning program, which attempts to explain an expert's behaviour by means of a proof tree constructed from meta-rules. When it is unable to construct such a tree, it assumes that one or more meta-rules failed to fire because one or more of their conditions were themselves unsatisfied. The conditions of meta-rules are a mixture of object and meta facts; ODYSSEUS makes the assumption that the failure of the meta-level proof is due to the lack of one or more necessary object level facts or rules, and adds to its object-level knowledge whatever facts are necessary to make the proof succeed. If there is

	Use of control information	Refinement of control information
Krust	Rule priorities affect proofs, and hence blame allocation.	Can refine a KB by changing rule priorities
Dolphin		Can add *object* conditions which make rule execution more efficient.
Teiresias	Meta-rules guide proof, but are not used in refinement. Rule models guide rule creation.	Rules are a source for rule models.
Odysseus	Meta-rules actually form the proof, and guide the creation of new rules.	Does not refine meta-rules directly, but adds object rules which change the behaviour of the meta-rules.

Table 2.1: The use of control information by various refinement systems

a choice of facts at this stage, ODYSSEUS uses a somewhat obscure set of heuristics to choose between them, depending on the nature of the failed meta-rule.

Thus, in terms of the framework presented earlier, ODYSSEUS effectively possesses only one refinement generation operator, together with obscure, and possibly domain-specific, refinement selection criteria. It is of interest, however, in that it is the only example of a system which carries out refinement of object-rules in order to change the behaviour of an expert system consisting of meta-rules.

Conclusions concerning the use of control information

Table 2.1 summarises the use of control information by the RSs discussed in this section. The right-hand column lists the ways in which the RSs can directly or indirectly manipulate control structures. It will be seen that the systems use and manipulate the control mechanism in a variety of different ways.

- KRUST directly manipulates the control mechanism (priority) of the KBS which it refines.
- ODYSSEUS indirectly manipulates an explicit control mechanism
- DOLPHIN indirectly manipulates an implicit control mechanism

- TEIRESIAS refines KBSs which use meta-rules, while neither learning from nor modifying these meta-rules. It does make use of a different kind of meta-rule, the rule models, but it constructs these itself from the KBS's object-rules.

2.4 Inductive Logic Programming (ILP)

The distinguishing feature of ILP is its ability to learn new predicates, given background knowledge and training examples. This overcomes two disadvantages of earlier inductive techniques such as ID3 3 (Quinlan 1986) and CN2 (Clark & Boswell 1991), which are unable to make use of background knowledge, and which are constrained by a fixed vocabulary of attributes. It also has the effect of reducing the number of examples needed to learn a concept.

The general inductive problem is as follows (Muggleton 1992). Given a set of observations O and background knowledge B, find a hypothesis H such that:

$$B \wedge H \vdash 0$$

The choice of H in this relation is severely under-constrained. Two ways of constraining H are to require it to be the least general hypothesis relative to B (Plotkin 1971), or to produce the maximum information compression; that is, the maximum reduction in the size of the theory.

The remainder of this section illustrates how the learning of rules can perform information compression in propositional logic. It goes on to show how propositional operators can be generalised to first-order logic, and that the generalised operators can be seen as reversing steps in a resolution proof. The creation of new rules by inverse resolution (IR) is a fundamental technique in ILP. Moreover, it is possible to construct a common framework for IR and Plotkin's relative least general generalisation (Muggleton 1992).

The following examples of the DUCE (Muggleton 1987) absorption and intra-construction operator illustrate both information compression and the learning of new predicates. Given the two rules:

$$a \ :- \ m, q, r, s.$$
$$b \ :- \ n, q, r, s.$$

the *absorption* operator identifies the common conditions q, r, s and creates the new rule

$$new \ :- \ q, r, s.$$

This allows the original theory can be re-written in a more compact form:

$$a \ :- \ m, new.$$
$$b \ :- \ n, new.$$
$$new \ :- \ q, r, s.$$

A second operator, intra-construction, works as follows. Given the two rules

$$a \ :- \ m, q$$
$$a \ :- \ n, q$$

it creates the new rule

$$new \ :- \ m$$
$$new \ :- \ n$$

thus allowing the theory to be rewritten

$$a \ :- \ new, q.$$
$$new \ :- \ m$$
$$new \ :- \ n$$

DUCE works with propositional logic, and employs 6 operators. MARVIN (Sammut & Bannerji 1986) extends the work into 1st order logic, but uses just one of DUCE's

operators: the absorption operator just illustrated. Muggleton & Buntine (1988) show that a number of DUCE's operators can be represented as inverting a step in a resolution proof, and in the CIGOL program build on this idea by implementing relational versions of DUCE's other operators.

The following example shows how the effect of the second of the DUCE operators just described, intra-construction, can be obtained by inverting two resolution steps. First a more general inverse resolution operator, the 'W' operator, is introduced. Then a special case of the application of this operator is shown to correspond to the intra-construction operator.

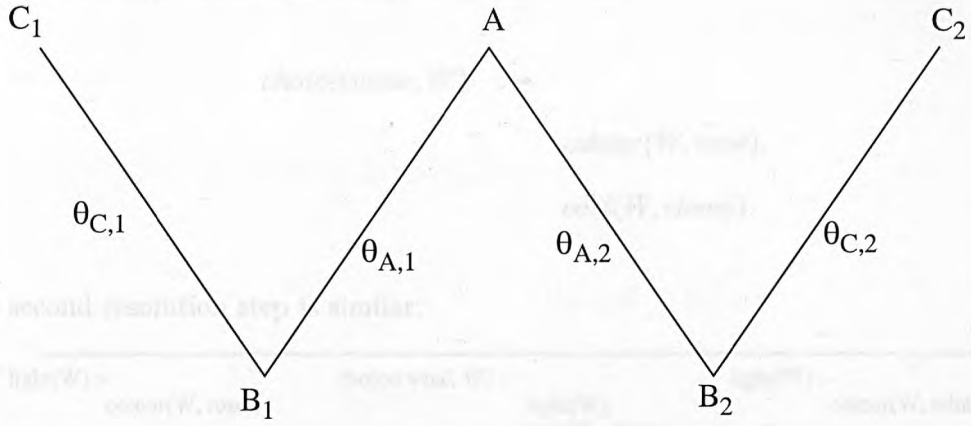

Figure 2.6: Two resolution steps with common clause A

Figure 2.6, taken from Muggleton (1992) illustrates the 'W' operator. The figure shows two resolution steps; C_1 and C_2 resolve on a common literal l within A to produce B_1 and B_2 respectively. The $\theta_{C,1}, \theta_{A,1} \ldots$ are the substitutions resulting from the resolution steps. The 'W' operator constructs the clauses A, C_1, C_2 given B_1 and B_2, so could be described as inverting the resolution steps.

There follows an example of the application of the 'W' operator (figure 2.7), which shows that it represents a 1st-order version of the DUCE intra-construction operator illustrated earlier. Here the common literal l is the condition *light(W)*. First one of the resolution steps is described. This step become clearer if the clauses involved are represented in disjunctive normal form. The first two clauses then become

$$light(W) \vee \neg colour(W, rosé)$$

$$choice(wine, W) \vee \neg light(W) \vee \neg cost(W, cheap)$$

which resolve to give

$$choice(wine, W) \vee \neg colour(W, rosé) \vee \neg cost(W, cheap)$$

This in turn may be represented as the rule

$$choice(wine, W) \; :- \; colour(W, rosé), \; cost(W, cheap).$$

The second resolution step is similar.

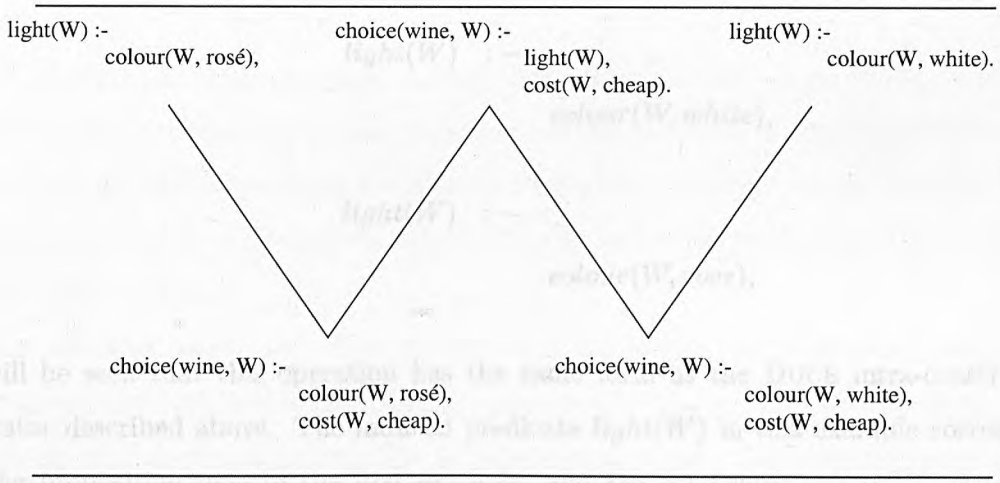

Figure 2.7: An example of intra-construction

The intra-construction operator works by inverting these resolution steps. It therefore starts with the two rules

$$choice(wine, W) \ :-$$
$$colour(W, rosé),$$
$$cost(W, cheap).$$

$$choice(wine, W) \ :-$$
$$colour(W, white),$$
$$cost(W, cheap).$$

and replaces them with the three rules

$$choice(wine, W) \ :-$$
$$light(W),$$
$$cost(W, cheap).$$

$$light(W) \ :-$$
$$colour(W, white),$$

$$light(W) \ :-$$
$$colour(W, rose),$$

It will be seen that this operation has the same form as the DUCE intra-construction operator described above. The induced predicate $light(W)$ in this example corresponds to the proposition *new* in the first example, and the predicates $colour(W, white)$ and $colour(W, rosé)$ correspond to the propositions m and n.

Currently ILP is developing as a field in its own right, and its techniques are not being used in refinement systems. However, they have been shown to be effective in real-world applications – domains include a satellite power supply and protein folding (Muggleton & Feng 1990), which suggests that they could be a useful alternative to the inductive operators currently employed by refinement systems.

2.5 Hybrid algorithms

The approach taken by the algorithms described in this section is to translate a symbolic theory into a neural net, train the net by back-propagation or some similar algorithm, and then translate the net back into a theory. Typically, it is the final translation phase that is most difficult, for reasons that will be explained below. In this section, the two hybrid algorithms KBANN (Towell, Shavlik & Noordewier 1990) and RAPTURE (Mahoney & Mooney 1993) are compared. Some comparisons are also drawn with PTR+ (Koppel, Segre & Feldman 1994).

2.5.1 Rule to net translation

The initial rule to net translation is similar in KBANN and RAPTURE. Figure 2.8 shows how the following two rules are translated in the two systems:

$$q \ :- \ a, b, c. \tag{2.1}$$

$$q \ :- \ d, e. \tag{2.2}$$

In KBANN, the unit bias for each intermediate node ($n\omega - \phi$, where n is the number of inputs) is chosen so that the intermediate node will be activated if and only if all its inputs are activated, thus representing a conjunctive condition. The bias for the conclusion node q is chosen so that it will be activated if either of its inputs are activated, thus representing a disjunctive condition.

On the other hand, the minimum and probabilistic-sum functions used by RAPTURE are non-standard for neural nets, but correspond more closely with one standard way of reasoning with uncertain information in rule-based systems. The probabilistic sum used for combining disjunctive conditions is expressed by the formula $x + y - xy$, where x and y are the two inputs, so that no thresholding is required, since the probabilistic sum provides the required non-linearity.

However, the most significant difference between KBANN and RAPTURE is that in KBANN, low-weighted links are added between every pair of nodes in successive layers which are not already connected. This permits KBANN to learn any necessary new relationships without modifying the net, by simply increasing the weight of the new links. On

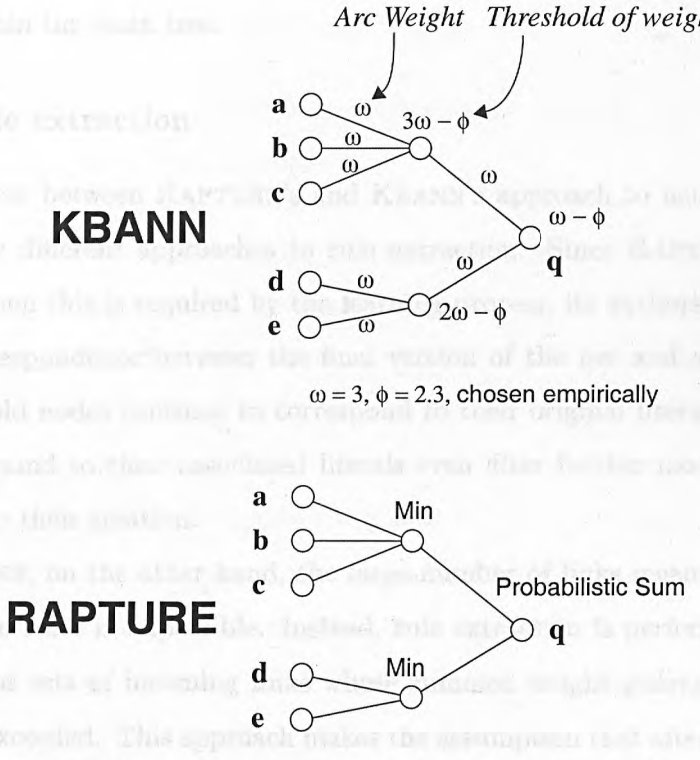

Figure 2.8: Rule to net translation in KBANN and RAPTURE

the other hand, RAPTURE does not create these extra links. It is therefore possible that the initial net is unable to learn the desired concept, in which case new nodes are added.

PTR+

PTR+ is included with the hybrid algorithms, although it does not strictly-speaking create a *neural* net, since it nonetheless uses a network representation. It represents a theory as a network, where each clause, and each literal in the head and body of a clause, is represented by a node, and arcs join each clause to each literal contained in the clause. Each arc has an associated value P which represents confidence that the edge is correct. Correct examples using an arc increase the value of P, and incorrect examples decrease it. If the value of P falls below a given threshold, this is regarded as evidence that the associated rule or condition needs repair.

A defect in PTR+ is that it cannot represent proofs of theories containing shared intermediate concepts, e.g., multiple uses of the same rule with different bindings. Brunk (1996) chapter 7 explains how this could be fixed by representing the theory as an and/or tree; a rule which is used in multiple contexts would appear several times as different

subtrees within the main tree.

2.5.2 Rule extraction

The differences between RAPTURE's and KBANN's approach to net construction require them to take different approaches to rule extraction. Since RAPTURE adds nodes and links only when this is required by the learning process, its authors believe that there is a direct correspondence between the final version of the net and a new theory — that is, that the old nodes continue to correspond to their original literals, and that any new nodes correspond to their associated literals even after further modifications to the net subsequent to their creation.

For KBANN, on the other hand, the large number of links means that a direct translation back to rules is impossible. Instead, rule extraction is performed by searching, at each node, for sets of incoming links whose summed weight guarantees that the node's bias will be exceeded. This approach makes the assumption that after the learning process is complete, all nodes will have activation level near 1 or near 0, and, as with RAPTURE, that the meaning of the nodes in terms of the original theory is not altered by the learning process. The drawbacks of this approach by KBANN, none of which is shared by RAPTURE, are that

- the translation back to rules is not deterministic,
- the behaviour of the rules obtained will not be identical to that of the net, and
- the complexity of the search is exponential in the number of input features.

Craven & Shavlik (1994) present an alternative approach to rule extraction from a neural net created by KBANN. This approach regards rule extraction as a learning task in which the target concept is the function computed by the network, and the attributes are the network's input features. Craven presents an algorithm for constructing a disjunctive normal form (DNF) expression corresponding to the classificatory behaviour of a neural net. In contrast to the previous approach, which requires extensive searching, this method largely treats the network as a black-box, so that it can approximate the network to an arbitrary degree of accuracy, and does not need to make any assumptions about the meanings of intermediate units.

2.6 Weaknesses in existing work

From the foregoing survey, the following problems, or at least areas which would benefit from further work, have been identified.

Handling multiple examples. All the programs surveyed deal with multiple examples in slightly different ways. The least efficient approach is to deal with examples one at a time, as KRUST and FOIL do; this approach has the added disadvantage that the accuracy of the final result depends on the order of example presentation. The solutions proposed for FOIL are the use of either beam search or check-pointing. Check-pointing would be applied whenever two or more steps scored roughly equally according to whatever heuristic was being used. If the step chosen lead to an unsatisfactory result, it would be possible to back-track to the checkpoint and choose one of the other options. Both these techniques could be applied equally well to refinement systems, though as yet neither has been.

The techniques could also be applied to systems that adopt a hill-climbing approach to the application of operators such as condition deletion or addition. The use of heuristics by such systems to choose the best condition at each stage makes them more efficient than those that are totally dependent on the order of example presentation, but the heuristics are not guaranteed to recommend the correct choice on every occasion, so the presence of techniques for back-tracking and making alternative choices could improve accuracy, at the cost of greater search.

Applicability. The systems described in this chapter are limited in their applicability. Each system is applicable to a single type of KBS only, usually one made up of PROLOG rules. Apart from CLIPS-R, they can only refine backward-chaining rules. This limits their applicability to industrial applications, where the use of forward-chaining rules is more common. Moreover, the systems have mainly been evaluated in relatively simple domains, where corruptions have been introduced manually. Consequently, their ability to refine industrial KBSs remains unproven.

Control mechanisms and proof quality. A conspicuous gap in the work described above, with the partial exceptions of DOLPHIN and ODYSSEUS, is the lack of attention paid to proof structures and proof quality; it is quite possible for a KB

to arrive at the correct conclusion by an incorrect chain of reasoning, but current refinement systems look only at a KB's conclusions. Moreover, patterns in proof structures could also provide further information to guide the refinement process.

Secondly, it has been noted that the refinement systems surveyed are generally unable to deal with systems containing forward-chaining, nor to take account of conflict resolution strategies. A partial exception is KRUST, which can refine rules where conflict is resolved by rule ordering.

These two areas of interest are closely related, since control and conflict resolution strategies (or meta-rules) guide the construction of proofs, and conversely any rule which attempts to critique a proof is by definition a meta-rule.

Chapter 3

The Tablet Formulation Application

This chapter introduces the Product Formulation Expert System (PFES) and the Tablet Formulation System (TFS) which is written in PFES. First it discusses formulation problems in general, and describes how knowledge-based systems may be employed to assist in formulation. It goes on to describe the PFES shell which is designed for writing formulation systems. It then introduces a specific formulation problem, that of tablet formulation. Finally, it describes TFS.

3.1 Formulation problems

Formulation usually begins with the presentation of some form of product specification of the requirements to be met, and ends with the generation of one or more formulations which meet the requirements (Alvey 1987).

The nature of the specification varies. It may be expressed in terms of performance levels to be met in a number of pre-defined tests. In other cases it will be much more vague; for example, that the product should be appropriate for a particular market sector. The formulation comprises a list of ingredients and their quantities, together with some process variables where this is appropriate; for example, the temperature at which the ingredients must be combined.

When a formulator attempts to solve a formulation problem by hand, he chooses

ingredients which he believes can be combined, taking into account interactions between them. He may need to add subsidiary ingredients to counter unwanted side-effects. He is likely to be guided in his choices by reference to previous formulations or historical test results.

This approach, of working from specification to formulation, describes many formulation processes, e.g., that of lubricating oils, agro-chemicals, plastics, dyes, explosives, glues, paints, food-stuffs, cleaning-agents and health-care products. In addition, other more varied tasks can also be modelled this way, such as the construction of investment portfolios, or project teams.

3.2 PFES

This section introduces the Product Formulation Expert System (PFES). It first describes the motivation for creating the system. It then gives an overview of PFES, describing its architecture, control structures, and knowledge representation mechanisms.

3.2.1 The role of knowledge-based systems in formulation

PFES was a Knowledge Based System Demonstrator project within the Alvey program, undertaken by a consortium consisting of Shell Research Ltd., Schering Agrochemicals Ltd., and Logica. The project investigated the applicability of KBSs in providing support for the formulation process. The following possible roles for such a system were envisaged; section 3.3.4 will show that the TFS application succeeded in performing many of these roles for the particular task of tablet formulation.

1. A decision support tool to assist the expert formulator.
2. A training aid for less experienced formulators.
3. A tool to assist experienced formulators in straightforward tasks, freeing them for more creative work.
4. A rationalisation tool for formulation knowledge and practice within an organisation.
5. A knowledge communication aid, for making a formulator's knowledge explicit and available to himself and other formulators.

6. A model building tool.

7. A database access tool.

The expected benefits of using a formulation system were as follows:

1. Elicitation of the formulation knowledge is made easier, since the framework into which the knowledge is structured is well-matched to formulation concepts.

2. The human formulator can remain directly involved in the project after implementation has started, again because the framework matches his approach to the problem.

3. Maintenance is made easier, since the framework makes it more straightforward to identify valid and appropriate modifications.

4. Explanations and traces generated by the system are comprehensible to the human formulator.

3.2.2 Overview of PFES

This section describes the PFES shell itself. PFES is a tool for writing formulation systems, which, as has been shown, can be applied to a wide range of problems. However, since the main reason for discussing PFES is to introduce the tablet formulation application, all the examples of PFES features, such as objects and rules, will be drawn from the tablet formulation domain. The following sections describe first how knowledge is represented within PFES, and then how that knowledge is used by tasks and rules.

The overall architecture of PFES is shown in figure 3.1. The formulation process is driven via the generation of a hierarchy of *tasks*, where each task represents some well-defined activity. The *control level* deals with the mechanics of running and passing control to these tasks. It is the *task level* where the actual formulation activity takes place, and where the hierarchy of tasks is generated dynamically. The purpose of the task hierarchy is to permit the formulation problem to be broken down into self-contained sub-problems. Tasks can plan about and directly manipulate only their immediate sub-tasks, which enforces the modular nature of the decomposition. Each task has an associated forward-chaining rule-set, whose action carries out the task.

An example of a TFS task is `Add Excipients`, which invokes the five sub-tasks

```
choose filler
choose binder
choose lubricant
choose disintegrant
choose surfactant
```

Finally, the *physical level* stores background knowledge about the formulation domain, and is accessed from the task level via a query interface.

An example of TFS background knowledge is the following database entry for the chemical mannitol.

```
(DEFOBJECT MANNITOL
    (YP 90.199997)
    (YP-FAST 161.0)
    (SRS 78.5)
    (SOLUBILITY 166)
    (IS-A FILLER)
    MIXES-WITH MAGNESIUM-CARBONATE CALCIUM-PHOSPHATE
               CALCIUM-DIHYDROGEN-PHOSPHATE))
```

3.2.3 Knowledge encoded in PFES systems

Three types of knowledge are encoded in PFES systems:

1. knowledge about how to build a formulation by adding components, setting levels, etc.;
2. knowledge about when and in which order the different aspects of formulation should be carried out;
3. knowledge about the components, processes and applications used in a formulation, together with their physical properties.

Physical knowledge is represented by objects, with properties represented by attributes. For example, figure 3.2 shows the PFES object CALCIUM-PHOSPHATE.

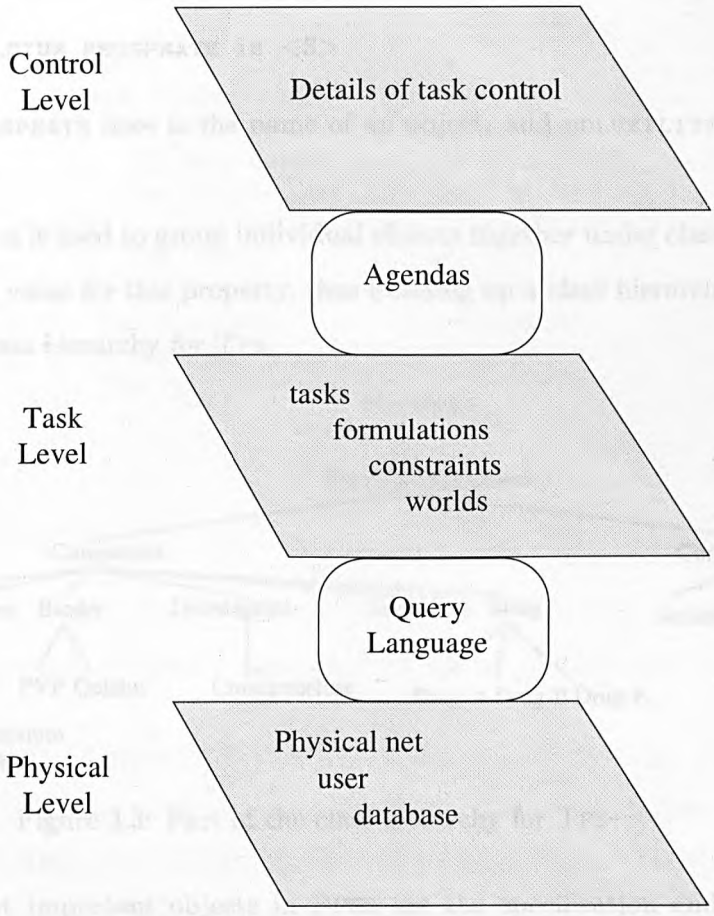

Figure 3.1: The architecture of PFES

CALCIUM-PHOSPHATE
IS-A: filler
YP: 957.2
YP-FAST: 957.2
SRS: 0
SOLUBILITY: 0
MIXES-WITH: MANNITOL MAIZE-STARCH LACTOSE MICROCRYSTALLINE-CELLULOSE
BONDING: INORGANIC

Figure 3.2: A PFES object

Note that although, in terms of the system architecture, this knowledge lies in the physical level, and is in fact stored in a separate database, a transparent interface is provided which allows PFES to treat the items in the database as objects at the task level. This means that a rule can determine, say, the solubility of calcium phosphate by means of the condition

SOLUBILITY of CALCIUM PHOSPHATE is <S>

where CALCIUM PHOSPHATE here is the name of an object, and SOLUBILITY is one of its attributes.

The attribute **is-a** is used to group individual objects together under class objects. All objects must have a value for this property, thus building up a class hierarchy. Figure 3.3 shows part of the class hierarchy for TFS.

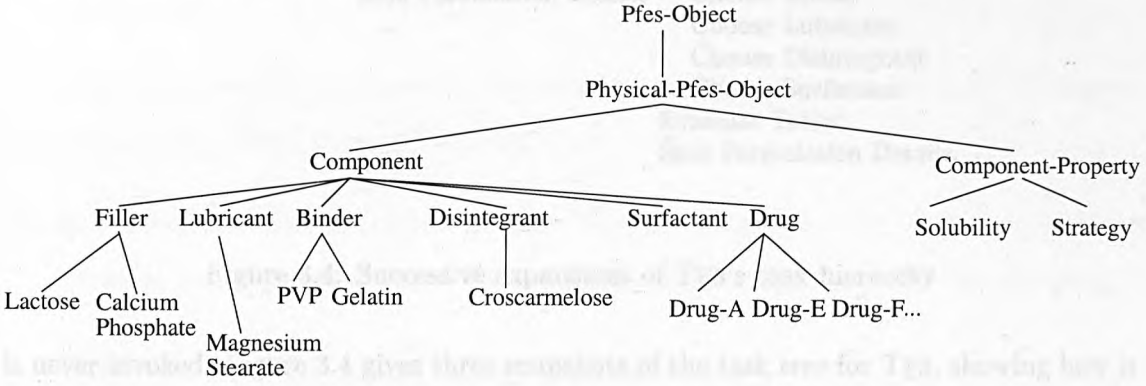

Figure 3.3: Part of the class hierarchy for TFS

Two of the most important objects in PFES are the specification and formulation objects. The specification object contains the requirements that must be satisfied by the formulation. As PFES executes, it constructs within the formulation object a description of the ingredients and quantities to satisfy the specification. In addition, the elements within PFES which manipulate the knowledge, viz., tasks, rule-sets and rules, are also objects, but as their role is rather different, they are described in the next section.

3.2.4 How the knowledge is used

The process of formulation can be broken down into self-contained activities or tasks. These are organised into a hierarchy under one primary task, which in the case of TFS is called *formulate*. Each task has a rule-set associated with it, and rules can themselves invoke tasks. As a result, PFES's task-tree is created dynamically each time PFES is run, and may vary depending on the initial specification. This contrasts with other systems, where the tasks are fixed and only the data varies. In the case of TFS, the task-tree does not vary greatly from case to case; possible variations arise when a particular excipient type such as surfactant is not required, so that the associated task, **choose surfactant**,

```
Formulate
  Initial Formulation
    Get Specification
    Add Excipients
    Evaluate Tablet
    Save Formulation Details
```

```
Formulate
  Initial Formulation
    Get Specification
      Get Drug Details
      Get Stability Data
        Check Stability Data
      Get Dose
      Set First Strategy
    Add Excipients
    Evaluate Tablet
    Save Formulation Details
```

```
Formulate
  Initial Formulation
    Get Specification
      Get Drug Details
      Get Stability Data
        Check Stability Data
      Get Dose
      Set First Strategy
    Add Excipients
      Choose Filler
      Choose Binder
      Choose Lubricant
      Choose Disintegrant
      Choose Surfactant
    Evaluate Tablet
    Save Formulation Details
```

Figure 3.4: Successive expansions of TFS's task hierarchy

is never invoked. Figure 3.4 gives three snapshots of the task tree for TFS, showing how it grows when TFS is run for one particular case. The initial task is **Formulate**. This creates a subtask **Initial Formulation**, which in turn creates four subtasks: **Get Specification**, **Add Excipients**, **Evaluate Tablet**, and **Save Formulation Details**.

Tasks can communicate with each other by means of *agendas*. An agenda is a named list. Once an agenda has been created, it is available to any of the tasks that run subsequently until either the agenda is explicitly deleted or the formulation process is complete. Information can be posted onto an agenda by one task for further tasks to use. Typically, agendas are used to pass data between routines that generate values and those that subsequently test or filter them.

3.2.5 Rule-sets and rules

Each PFES task has an associated rule-set, consisting of a number of *if-then* rules. This section first describes the types of conditions and conclusions which may appear in PFES rules, and then describes PFES's control and conflict-resolution strategies.

Rule elements

PFES rule elements, that is, rule conditions or conclusions, may be divided into the following three classes (Logica 1988).

- Object-Attribute-Value triples
- Agenda operations
- Lisp forms

Some examples of each type of rule element are now presented. In these examples, items in angle brackets, such as <WEIGHT>, are variables.

Object-Attribute-Value triples have the same meaning as the OAV-triples which KRUST reasons about (section 4.1.2). When they occur as conditions, they take the form

```
<ATTRIBUTE> has value <VALUE> in <OBJECT>
```

When they occur as actions, they take the form

```
set the value of <ATTRIBUTE> in <OBJECT> to be <VALUE>
```

Agenda operations read and write items from and to an agenda. Here are two typical agenda operations.

```
add <VALUE> to <END> of <AGENDA>
```

where <END> may be `top` or `bottom`. This adds <VALUE> to the top or the bottom respectively of <AGENDA>.

```
add <VALUE> <POSITION> <VALUE2> on <AGENDA>
```

where <POSITION> may be `before` or `after`. This adds <VALUE> to the agenda, either directly before or directly after <VALUE2>, which must already be on the agenda. There exist equivalent reading operations, which read items from the top or bottom of the agenda, or from next to some named item.

Comparisons take the form

<VALUE> is <COMPARATOR> <EXP>

<COMPARATOR> may take any of the values $>, \geq, <, \leq, =$.

Lisp forms A limited number of Lisp functions are available. These are arithmetic operators, `max` and `min` functions, and truncation operators.

Interface operations read data from the user, and display information on the screen. These are not described here, since they are irrelevant to the automated refinement of PFES applications.

Figure 3.5 shows a typical rule, first in its LISP representation, then as it is displayed in the PFES graphical interface.

Rule execution

When a task is scheduled, the interpreter orders the rules in the associated rule-set firstly by priority and second by specificity. The interpreter then attempts to fire each rule in turn. If a rule's conditions are satisfied, and the exceptions, if any, are not, then the rule fires, and the rule's conclusion is executed. If at any point a condition fails, the interpreter searches the list of rules in the same way as before. Some conditions may succeed multiple times with different bindings; the interpreter will repeatedly retry successful rules as long as the bindings of at least one variable continues to change. For example, the first condition of the rule `GET-STABLE-FILLERS` succeeds multiple times as it reads successive items off an agenda.

```
Rule GET-STABLE-FILLERS
If
    <FILLER> is on STABLE-FILLER-AGENDA
Then
    add <FILLER> to bottom of FILLER-AGENDA
```

The rule will therefore fire once for each item on the `FILLER-AGENDA`.

```
(DEFOBJECTD 1ST-GUESS-WEIGHT
    (IS-A RULE)
    (PRIORITY 0)
    (MEMBER-OF-RULESET GET-DOSE-ACTION-RULESET)
    (TASK GET-DOSE)
    (ROLE ACTION)
    (CONDITIONS
        (F-S-ATTRIBUTE-HAS-VALUE FORMULATION DRUG <DRUG> <>)
        (F-S-ATTRIBUTE-HAS-VALUE FORMULATION <DRUG> <DOSE> <>))
    (ACTIONS
        (LISP-TO-VALUE (ROUND-TO-NEAREST
            (/ <DOSE> (+ 0.1 (* 0.00221 <DOSE>))) 5) <WEIGHT>)
    (SET-ATTRIBUTE-VALUE SPECIFICATION TARGET-TABLET-WEIGHT
            <WEIGHT> < >))
    (EXCEPTIONS)
    (BECAUSE "weight = 100 * dose /(0.221 * dose + 10) eqn(1)"))

Rule 1ST-GUESS-WEIGHT
IF
    DRUG has value <DRUG> in the FORMULATION
    <DRUG> has value <DOSE> in the FORMULATION
THEN
    set <WEIGHT> to the result of (ROUND-TO-NEAREST
            (/ <DOSE> (+ 0.1 (* 0.00221 <DOSE>))) 5)
    set the value of TARGET-TABLET-WEIGHT in the SPECIFICATION to be <WEIGHT>
BECAUSE
    "weight = 100 * dose /(0.221 * dose + 10) eqn(1)"))
```

Figure 3.5: A TFS rule, first in internal form, then in pretty-printed form

3.3 Tablet formulation and the TFS system

3.3.1 The tablet formulation problem

Well over half of all the medicines used in the UK are in the form of compressed tablets (Rowe 1993). This form is convenient both for the manufacturer and the patient. When tablets are manufactured, a number of inert components known as excipients are added to impart necessary properties, such as strength and stability, to the tablet. The following types of excipients are commonly required.

Filler: provides the bulk of the tablet, which would otherwise be too small to be easily handled.

Binder: prevents the tablet breaking or crumbling during storage.

Lubricant: prevents the tablet from sticking to punches and dies when being stamped.

Disintegrant: causes the tablet to break down after being swallowed by the patient.

Surfactant: is a wetting agent which may be needed if the tablet is otherwise water-resistant.

The task of tablet formulation is normally performed by a limited number of experts with specific knowledge and often years of experience. Their knowledge is not easily documented, so that the experts often spend considerable time in training new personnel. Other disadvantages of the reliance on experts are that personal preferences can result in inconsistencies of approach, and that irreplaceable knowledge may be lost when experts retire. An expert system for tablet formulation would solve these problems by providing a means of capturing available knowledge and implementing a consistent approach, and could be updated in the light of new practices. This was the motivation for building the TFS system, which has become one of the few formulation KBSs in regular commercial use (Frank, Rupprecht & Schmelmer 1997).

3.3.2 The TFS system

TFS's knowledge is encoded in PFES tasks, rules and objects. A typical object, task tree and rule have already been presented (figures 3.2, 3.4 and 3.5). The objects, which

represent chemical properties of excipients and drugs, are stored in a database which may be accessed by the rules.

From the user's point of view, the operation of TFS proceeds as follows. The user provides a drug's name and its desired dosage. TFS then uses these values, together with other information read from a database, to generate the specification; that is, necessary properties of the tablet to be formulated, such as its weight, size, and various physical properties. TFS then generates a formulation consisting of the most appropriate material from each excipient type, and the quantity of each required. Figure 3.6 shows an example of the user input, specification and resulting formulation for one particular example.

User Input:

Drug: Drug-A
Dose: 60 mg
No of fillers: 2

Specification:

full-stability: Yes
drug-filler–concentration: 0.9
minimum-tablet-weight: 100mg
maximum-tablet-weight: 800mg
target-tablet-weight: 260mg
start-strategy: strategy-A
filler-concentration: 66.9%
typical-disintegrant: Maize-starch
disintegrant-concentration: 0.05
tablet-weight: 252.2mg
total-concentration: 97%
tablet-diameter: 8.73mm
... various other properties...

Formulation:

Tablet weight: 250mg
Fillers: Lactose 66.7%,
Calcium phosphate 2.4%
Binder: Gelatin 4.1%
Lubricant: Stearic acid 1.0%
Disintegrant: Croscarmelose 2.1%

Figure 3.6: An example of user input, specification and formulation

3.3.3 History of TFS

The first version of TFS was created by Zeneca in 1988 and labelled TFS-1A. This has been described by its users as a prototype, but still a useful system. A later development resulting from some internal modifications and bug fixes, and in use for two years, was TFS-1B. The last version, TFS-2, incorporated a number of further modifications including

one fundamental change, discussed below, and was completed in 1991.

The fundamental change resulted from the introduction of three categories of excipients, where category 1 included the most preferred excipients, and category 3 the least preferred. Formulators were required to use excipients in category 1 if possible, only picking excipients from the later categories if this was necessary to create an acceptable formulation. The purposes of the introduction of these categories were:

1. Harmonisation of inventories.
2. Cost-saving.
3. Greater consistency of products. Natural products, which may vary slightly from batch to batch, are relegated to category 3, and so are rarely used.

As a result, the TFS rule-base had to be modified to implement this policy. This was a significant task, and required several man-months of effort.

3.3.4 Benefits of TFS

The creation of TFS has provided Zeneca with the following benefits (Turner 1991).

- The creation of a permanent database of formulation expertise.
- The provision of a training aid for both novice and experienced formulators.
- The guarantee of a consistent approach to formulation.
- The provision of a common starting point for discussing and managing changes in formulation practice.
- Acceleration of the formulation process, and reduction in usage of materials. (When TFS is used for formulation, the number of different formulations which need to be tested in the laboratory is reduced).
- The release of experienced staff for more innovative work, rather than the creation of relatively standard formulations.
- Identification of critical areas of formulation which required further research or rationalisation.

These compare well with the benefits envisaged by the PFES project for an automated formulation system.

3.4 Summary

This chapter has shown that design and formulation are hard problems, and have to be tackled in a different way from problems of diagnosis and classification. The PFES shell, which has been created to solve formulation problems, has a number of features which distinguish it from other general purpose expert system shells; in particular, its dynamic task structure, and its use of agendas to exchange information. A particular PFES application, TFS, has proved very successful, but the paucity of other automated tablet formulation systems indicates the difficulty of this problem. These features show that the automated refinement of TFS is a demanding task for a refinement tool, and will constitute a significant contribution to the field.

Chapter 4

KRUST

This chapter describes the knowledge base refinement system KRUST. It first addresses the history and development of KRUST, and explains which version is principally to be discussed. The remainder of the chapter is devoted to an account of how KRUST works. Section 4.1 describes KRUST's knowledge representation: the kind of rule conditions and conclusions it is able to represent and reason about. Section 4.2 describes how KRUST obtains information about the behaviour of the KBS which it is refining. Section 4.3 presents an overview of the operation of KRUST. This operation is divided into a number of stages, each of which is described in greater detail in subsequent sections.

Once the description of KRUST is complete, there follows a brief discussion on the nature and requirements of a generic refinement tool, and some conclusions about the strengths and weaknesses of KRUST compared with other refinement tools. These last two sections provide motivation for my own work on applying KRUST to TFS, described in chapter 5.

This chapter discusses the original version of KRUST which is described in more detail in Craw (1991). This version was applicable to PROLOG KBs only. Since then, my work and that of other researchers has developed KRUST into a generic refinement tool, applicable to TFS, CLIPS and POWERMODEL (Palmer 1995). I shall therefore indicate here which features of KRUST's knowledge representation and operators made the program amenable to development into a generic tool. Chapter 5 describes my work on the application of KRUST to PFES, and makes clear the extent of TFS's influence on the tool in its present form.

4.1 Knowledge representation

This section explains why KRUST needs to represent the knowledge in a KB, and what kinds of knowledge can be represented.

4.1.1 The knowledge skeleton

The knowledge skeleton is KRUST's internal representation of the rules and facts in a knowledge base. The knowledge skeleton is used to augment the information available to KRUST about *actual* rule firing behaviour, by allowing it to reason about the consequences of *changes* to the rules. This section explains the difference between these information sources, and why both are necessary.

First of all, KRUST needs to know which rules in a KBS actually fired, and which did not. This information can be determined directly from the KBS. The information is used to determine that some rules fired which should not have done, and *vice versa.* Since the information is derived from the KBS itself, there is no requirement for a knowledge skeleton at this point.

However, KRUST next needs to determine how to change the rules in order to produce correct behaviour. It has to reason about chains of rules, where a change to one rule will affect the behaviour of others. Communication with the KBS is no use here, since KRUST needs to reason about interactions between rules which did not occur in the original running of the KB. It is at this point that the knowledge skeleton is used. The purpose of constructing the knowledge skeleton is therefore *not* to run a simulation of the shell, but rather to provide the extra information needed to reason about the possible effects of changes to the rule-base. KRUST can determine from the KBS which rules *actually* fired, but during refinement it must also reason about *potential* rule-firing behaviour. For example, it may need to determine which rules in the KB have a conclusion which matches a given rule condition. This kind of information has to be derived from the knowledge skeleton.

4.1.2 Rule elements

In the original KRUST, the only permitted rule element (rule condition or conclusion) is the object-attribute-value (OAV) triple; for example, **colour of wine is red**, where **colour**

is the attribute, **wine** the object, and **red** the value. However, KRUST's representation also allows it to associate data-types and constraints with each OAV triple, so this is not as restrictive as it might first appear. The following attribute types are permitted:

Discrete: A finite unordered set, such as {red, white, rosé}.

Linear: A finite or infinite ordered set, such as the integers, or the set {small, medium, large}

Tree: A partially-ordered set, where $X > Y$ if X is an ancestor of Y in the associated tree.

The constraints that may be represented depend on the attribute types. The only possible constraint on a discrete attribute is equality. For example, **colour of wine is red**. On the other hand, a linear attribute can be upper-bounded, lower-bounded, or both; thus KRUST can represent conditions such as **cost of meal < 30** or **20 < cost of meal < 40**. Finally, the only constraints that can be placed on a tree attribute are upper bounds. For example, the constraint **origin of wine is europe** (see figure 4.1) requires the origin of the wine to lie on or below the node labelled "Europe".

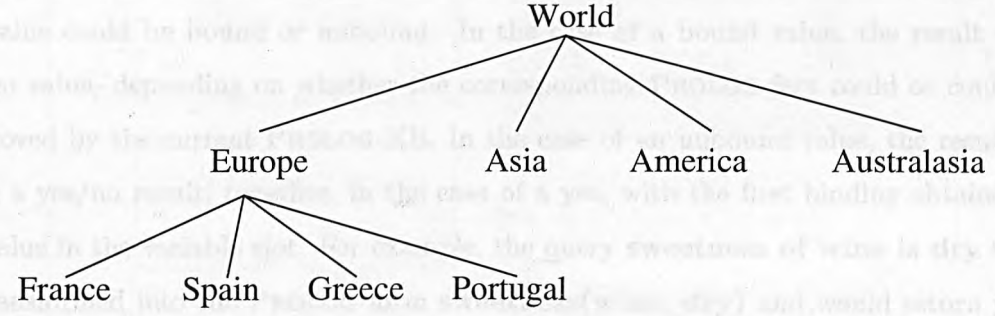

Figure 4.1: A tree-structured attribute

The type declarations are included in meta-knowledge which may be associated with any KB to which KRUST is to be applied; the actual values of the bounds appear in the rule conditions themselves. Note that for integer and real-valued attributes, the upper and lower bounds are simply an alternative way of expressing conditions which could more conventionally be expressed by comparisons of the form $Val < Bound$, or $Bound_1 < Val < Bound_2$. However, the ability to reason about tree-structured attributes, and ordered sets other than integers and reals, is a distinguishing feature of KRUST.

The usefulness of this representation for a refinement tool lies in two of its features:

- it permits the tool to select the minimal change to an OAV triple which will give the desired behaviour;
- it permits the tool to make use of any available background knowledge about properties of the attributes.

4.2 Communication with the KB

KRUST seeks to modify the rules in the KB so that they will behave differently, thereby generating some new conclusion. It therefore needs to know which rules *currently* fire. More precisely, it requires answers to the following queries.

1. Is rule R satisfied, and what variable bindings result?
2. If rule R is not satisfied, which condition(s) cause it to fail?

In the original version of KRUST, this information was obtained via a query sent from KRUST to the PROLOG process, an approach which relies on PROLOG's ability to instantiate variables in an arbitrary query. The query took the form of an OAV triple, in which the value could be bound or unbound. In the case of a bound value, the result was a yes/no value, depending on whether the corresponding PROLOG fact could or could not be proved by the current PROLOG KB. In the case of an unbound value, the result was again a yes/no result, together, in the case of a yes, with the first binding obtained for the value in the variable slot. For example, the query **sweetness of wine is dry** would be transformed into the PROLOG form **sweetness(wine, dry)** and would return **yes** if this result could be proved, **no** otherwise. The query **sweetness of wine is _S**[1] would be transformed into **sweetness(wine, _S)** and might return **yes**, with **_S** bound to **dry**, say, or **no**, with no bindings.

KRUST can combine such queries about individual conditions to determine whether a rule as a whole is satisfied. To do so, KRUST submits each of its conditions in turn as queries. When a query binds a variable, that binding is retained if the same variable occurs in later queries. Thus the sequence of queries determines whether the rule as a whole can fire, and if not, which conditions caused it to fail.

[1] In KRUST, variables start with an underscore.

4.3 The operation of KRUST

Figure 4.2 outlines how KRUST operates. Its input is a KBS, and a refinement example, which will drive the refinement process. The refinement example consists of the input data describing a particular problem, together with the desired output for that problem, usually obtained from an expert. KRUST's first action is to run the KBS for the given input. If the output matches the expert's, then no refinement is required. Otherwise KRUST will attempt to change the rules in the KBS so that it will solve the refinement example correctly.

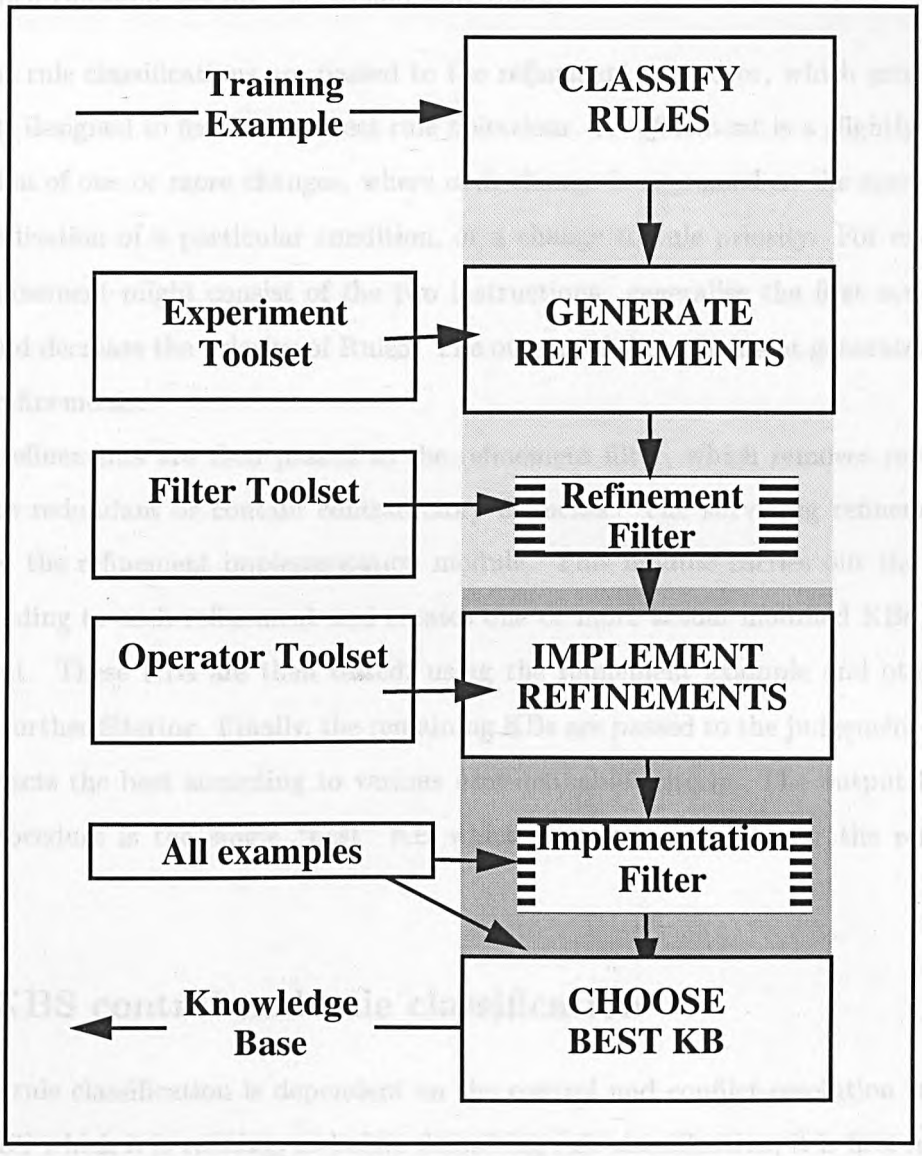

Figure 4.2: The operation of KRUST

There follows an outline of the various steps in the refinement procedure. These steps are then described in more detail in sections 4.4 to 4.8. The first step is rule classification, which is a specialised form of blame allocation; it consists of describing the incorrect behaviour of a KBS's *end rules*. The end rules are those rules which can occur at the root of the KBS's proof tree, and are directly responsible for the KBS's conclusion. Roughly-speaking, classification consists of determining

- which rules fired but should not have done, and
- which rules did not fire and should have done.

These rule classifications are passed to the refinement generator, which generates refinements designed to fix the incorrect rule behaviour. A *refinement* is a slightly abstract description of one or more changes, where each change is expressed as the specialisation or generalisation of a particular condition, or a change to rule priority. For example, a single refinement might consist of the two instructions: generalise the first condition of Rule1, and decrease the priority of Rule5. The output of the refinement generator is a list of such refinements.

The refinements are then passed to the refinement filter, which removes refinements which are redundant or contain contradictory elements. The surviving refinements are passed to the refinement implementation module. This module carries out the changes corresponding to each refinement and creates one or more actual modified KBs for each refinement. These KBs are then tested, using the refinement example and others, and undergo further filtering. Finally, the remaining KBs are passed to the judgement module, which selects the best according to various user-definable criteria. The output from the whole procedure is the single "best" KB which performs correctly for the refinement example.

4.4 KBS control and rule classification

KRUST's rule classification is dependent on the control and conflict-resolution strategies of the KBS which it is refining, so before describing rule classification, it is first necessary to discuss briefly these properties of a PROLOG KBS.

4.4.1 Control

The control-strategy of a PROLOG KBS is depth-first backward-chaining. The need for a conflict-resolution strategy arises when several rules exist whose conclusions unify with the rule condition which PROLOG is currently trying to prove. These rules are known as the *conflict set.* PROLOG's strategy is to select the rules from the conflict set in the order in which they appear in the KB; if the first rule does not fire, then the second one is tried, and so on. This is a special case of the numerical priority assigned to rules in systems like Clips; the behaviour of PROLOG rules could be modelled by numerical priorities if we assign a priority of 0 to the last rule in the KB, a priority of 1 to the last rule but one, and so on.

Another property of a KBS which must be considered is the criterion for inference to stop. In PROLOG it is possible to submit a query, receive a successful response, and then force back-tracking to get further responses. However, KRUST regards the first response only as the PROLOG KBS's "output". As a result, the relative priorities of the end-rules in the KB are of particular importance. Recall that the end-rules are those that are directly responsible for the system's conclusion, or in the case of a PROLOG database, are those that match the query. The importance of end-rule priority may be illustrated by means of the following simple KB.

$$p(a) \ :- \ q. \tag{4.1}$$

$$p(b) \ :- \ r. \tag{4.2}$$

$$q \ :- \ x. \tag{4.3}$$

$$q \ :- \ y. \tag{4.4}$$

$$r \ :- \ z. \tag{4.5}$$

$$y. \tag{4.6}$$

$$z. \tag{4.7}$$

Given a query *?- p(X).* the initial conflict set consists of rules 4.1 and 4.2. PROLOG considers rule 4.1 first, so attempts to prove condition q. The conflict set for q consists of rules 4.3 and 4.4. Rule 4.3 can not be satisfied, since x can not be proved, but rule 4.4

can. Hence q is proved, rule 4.1 fires, and the value $p(a)$ is returned. Because inference stops as soon as the first end-rule fires, the lower-priority end-rule, rule 4.2, never fires, even though its condition can be satisfied.

4.4.2 Rule classification

The purpose of KRUST's rule classification is to describe the incorrect behaviour of a KBS's end rules. The behaviour of an end-rule in a backward-chaining KBS's rules is determined by two factors: whether the rule's conditions are satisfied, and the relative priority of other rules whose conditions are also satisfied. It is these two factors which are taken into account by KRUST's classification step.

In the following discussion, a rule is *satisfied* if its conditions are provable. Let E be the expert's conclusion, and S the system's conclusion for a particular refinement example. We assume that $S \neq E$, or there is no need for refinement. Then the rule classes are defined as follows.

The Error-Causing rule is the satisfied rule which wins the conflict resolution, and concludes S.

A target rule is one which would conclude E if it fired.

Target rules may be further divided as follows, depending on the reasons they failed to fire.

No-Fire rules have a high enough priority to fire, but their conditions are not satisfied.

Can-Fire rules are those whose conditions are satisfied, but failed to fire because of their low priority.

NoCan-Fire rules have too low priority to fire, and in addition their conditions are not satisfied.

Finally, the class **Potentially-Error-Causing** describes end rules which do not conclude E, whose conclusions are satisfied, and have a lower priority than the error-causing rule, but a higher priority than one or more Can-Fire or NoCan-Fire rules. The significance of this class is that Potentially-Error-Causing rules must be prevented from firing if they preclude the firing of a target rule.

4.5 Refinement generation

The refinement generator builds a set of refinements designed to correct the erroneous behaviour of the end rules identified by the rule classifier. Its output is a list of refinements, where each refinement is a conjunction of rule changes and rule priority changes. Roughly speaking, each refinement is designed to prevent the Error-Causing and Potentially-Error-Causing rules from firing, and to allow one of the target rules to fire instead. More precisely, the rule changes for the various classes of rules are shown in figure 4.3, taken from Craw (1991), p. 60, with slightly modified terminology. In the following discussion,

- to *enable* a rule means to cause its conditions to be satisfied so that it will fire if its priority is sufficiently high, and
- to *allow* a rule means to modify both its conditions and its priority as necessary so that it *will* fire.

The meanings of *disable* and *disallow* are defined similarly. (To disallow a rule means to prevent it from firing *either* by causing one of its conditions not to be satisfied, *or* by decreasing its priority). It is also useful to talk about enabling or disabling an individual rule *condition.*

Disabling a rule R can be carried out in one of two ways: by strengthening *any one* condition C of R until it is no longer satisfied, or by simultaneously disabling *all* satisfied rules which conclude C. The process of enabling a rule is complementary to this. A rule can be enabled by simultaneously enabling *all* of its unsatisfied conditions C_i. A condition C_i can be enabled either by weakening it until it is satisfied, or by enabling *any one* rule which concludes C_i. Note that disabling and enabling, which are solely concerned with the satisfiability of rule conditions, are defined recursively, but that operations on rule priority are confined solely to the end rules. This reflects the importance of the priority of end-rules in PROLOG as explained in section 4.4.1. Since KBS inference stops as soon as one end-rule fires, these rules can be prevented from firing by lowering their priority, but that this is not the case for rules lower down the proof tree.

We next consider how the refinements in figure 4.3 need to be combined in order to correct the behaviour of a rule-set as a whole. For example, to cause a NoCan-Fire rule to fire, it is necessary to weaken its premises, but it might also be necessary at the same

Class	Change	Mechanism for change
No-Fire rule	Allow	by weakening the premises
Error-Causing rule	Disallow	by strengthening a premise OR by decreasing its priority
Can-Fire rule	Allow	by increasing its priority
NoCan-Fire rule	Allow	by weakening its premises AND by increasing its priority

Figure 4.3: Refinements for the rule classes

time to disable Potentially Error-Causing rules.

When describing combinations of refinements, it is helpful to introduce the concept of the *cartesian conjunction* of a set of refinements, which is related to the concept of a Cartesian product in set theory. Suppose Ref_i are sets of refinements. Then the *cartesian conjunction* of Ref_i is the set of all conjuncts $\wedge_i r_{ij}$, where $r_{ij} \in Ref_i$. The need for such a combination may be illustrated by means of the following simple example. Suppose it is necessary at the same time to disable rule R_1 and to enable rule R_2. Let Ref_1 be the refinements which disable R_1, and let Ref_2 be the refinements which enable R_2. Then a refinement which does both may be obtained by picking any one refinement from Ref_1 and combining it with any one refinement from Ref_2. In other words, the set of refinements which disable R_1 and enable R_2 is the cartesian conjunction $r_{1j} \wedge r_{2k}$, where $r_{mn} \in Ref_m$

The following table lists for each class of end rule the refinements needed to correct its behaviour. In addition, there is a single refinement at the end, "create a new rule", which does not correspond to any end rule. The table is taken from Craw (1991), pp. 62–64, slightly modified. The final set of refinements generated by KRUST is the disjunction of the refinements generated for each target and error-causing rule.

No-Fire: For each unsatisfied condition C_i, either weaken the condition until it is satisfied, or enable any one rule whose conclusion satisfies the existing condition. This generates sets of refinements Ref_i. Return the cartesian conjunction of Ref_i.

Error-Causing: Change the conclusion of the Error-Causing rule to E.

Can-Fire: Generate the cartesian conjunction of the following three sets of refinements:

- increase the priority of the Can-Fire rule just above the highest priority Can-Fire rule,
- disable the Error-Causing rule or decrease its priority below the Can-Fire rule,
- disable or decrease the priority of all Potentially Error-Causing rules with priorities between the Error-Causing rule and the Can-Fire rule.

Then add the final single refinement to the above conjunction: increase the priority of the Can-Fire rule just above the Error-Causing rule.

NoCan-Fire: Generate two sets of refinements: one for the rule treated as a No-Fire rule, and one for the rule treated as a Can-Fire rule. Return the cartesian conjunction of these two sets.

(No associated rule type): Build a new rule

$$F_1 \wedge F_2 \wedge \ldots \wedge F_k \rightarrow E$$

where the F_i are the known facts for the refinement case.

It will be seen that the procedure for generating a new rule produces the most specific rule possible which will apply to the refinement case. Such a rule is not normally a very useful one. KRUST was unable to do any better because it lacked inductive operators which could use multiple examples to learn a more general rule. This lack also prevented KRUST from learning new rule conditions. Chapter 6 describes how I have to some extent remedied this deficiency by adding various inductive operators.

4.5.1 Refinement filtering

At this point, KRUST removes those refinements which, for various reasons, are unlikely to be useful. It bases its decisions on the nature of the refinements themselves considered in isolation, not on the effectiveness of the refined KBs, which have not yet been generated. There are three types of refinement filter.

Inconsistency and redundancy removal. An *inconsistent* refinement is one containing two conflicting rule-changes, such as specialising and generalising the same rule condition. A *redundant* refinement is one whose changes are a strict super-set of some other refinement's changes.

Meta-knowledge. This filter removes refinements to rules believed to be "good", provided some measure of goodness exists. An example of a goodness measure is that used by SEEK (Ginsberg 1988a), which assigns a **gen_weight** and a **spec_weight** to each rule, according to the number of generalisation and specialisation refinements generated for that rule. Rules with weights below a certain threshold are considered "good".

Heuristic. This class of filters was intended to permit the introduction of any other criteria for refinement deletion not covered by the previous two classes. For example, domain-specific knowledge might be introduced at this point. However, the only heuristic filter currently available is a non-domain-specific one which prefers simple refinements to complex ones. It sorts refinements into order depending on their complexity (number of changes), and if the number of refinements is greater than a user-defined threshold, it deletes the most complex refinements.

4.6 Refinement implementation

The output of the refinement generation process is a set of general and slightly abstract instructions for making rule changes: for example, generalise condition C in rule R1, move rule R2 above rule R3, change the conclusion of rule R4. The purpose of the refinement implementation stage is to turn these instructions into actual rule changes. There may be many ways of implementing any particular instruction; for example, to generalise a condition involving comparison with a threshold, it is possible to change the threshold, change the comparison operator, or remove the condition altogether. Moreover, the possible changes depend on the nature of the condition itself. Consequently, KRUST uses a collection of *refinement operators* where each operator is associated with one or more refinement types, and where each operator may be restricted to certain condition types (figure 4.4). This tool-set is a feature of KRUST which is naturally extensible to

new shells, since new operators can be added to apply the existing refinements to new rule elements.

To Generalise:	Remove Condition
	Adjust Value
	Adjust Operator
	...
To Specialise:	Delete Rule
	Adjust Value
	Adjust Operator
	...
To Allow:	Increase Priority
	...
...	

Figure 4.4: The KRUST refinement operator toolset

The attribute types described in section 4.1.2 permit KRUST to use a finer generalising and specialising mechanism than the rule/condition deletion/addition approach of some older refinement systems. Instead, KRUST changes the value in OAV conditions just enough to have the desired effect for the particular refinement task. If no suitable change can be found, then KRUST is still able to apply the remove condition operator (the ultimate generalisation) or the delete rule operator (the ultimate specialisation).

4.6.1 Changing conditions

This section describes how KRUST refines a comparison by altering the bound within the OAV condition. KRUST first queries the KB to determine the value actually taken by the attribute when the KBS is applied to the refinement case. Then, to *specialise* a condition, it adjusts the bound in the rule condition just far enough to *prevent* the condition from being satisfied; alternatively, to *generalise* the condition, it adjusts the bound just far enough to *allow* the condition to be satisfied. For example, suppose KRUST wishes to specialise the condition **cost of meal** $<$ **30**. It queries the KB to determine that the value of **cost of meal** for the refinement case is 25. It makes the minimal change to the condition which will prevent it being satisfied by the refinement case, so generating the modified condition **cost of meal** $<$ **25**.

This approach can be applied unambiguously to single-bounded numerical and ordered

attributes. A few special cases remain to be dealt with.

Double-bounded attributes. KRUST adjusts whichever of the two bounds requires the least change.

Tree-valued attributes. KRUST generalises a tree-valued condition by moving to the least general ancestor node that causes the condition to be satisfied. For example, consider the geographical hierarchy shown earlier, and reproduced here in figure 4.5. Suppose the condition **origin of wine = France** fails, because the origin of wine is Greece. KRUST will generalise the bound to be the least general ancestor of France and Greece, viz., Europe.

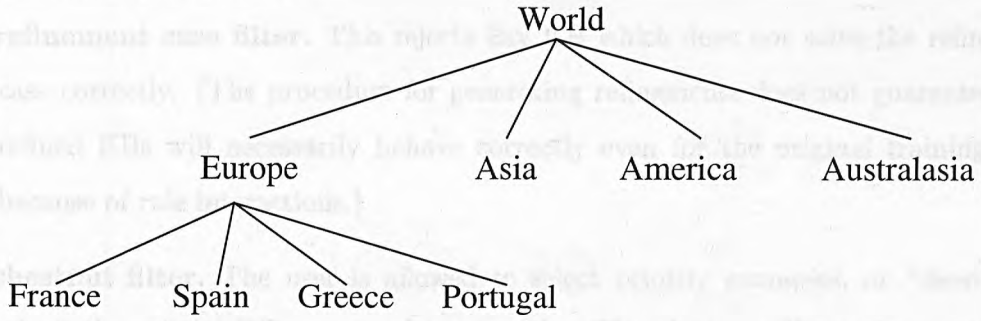

Figure 4.5: A tree-structured attribute

Specialisation is harder, since there may be no unique minimal specialisation which causes a condition to be unsatisfied. For example, suppose the condition **origin of wine = Europe** succeeds, with the origin being France. Then three possible specialisations exists (to Spain, Greece or Portugal), and KRUST will generate one replacement rule for each specialisation. This is equivalent to replacing the original condition with the disjunction **origin of wine = Spain OR origin of wine = Greece OR origin of wine = Portugal.**

Discrete attributes Since discrete attributes have no order or other structure, a condition on a discrete attribute can be specialised only by replacing it with the condition **False**, or equivalently, deleting the entire rule, which is what KRUST actually does. The condition is generalised by generating a new rule for each possible value of the attribute which is satisfied. (This is a complementary operation to specialising a tree-structured attribute). For example, suppose possible values of **Colour** are **red, white, rosé**. Then suppose KRUST is required to generalise the condition **colour of**

wine = red. KRUST first determines for which values of _C the condition **colour of wine = _C** is satisfied. If these are **white** and **rosé**, KRUST replaces the original rule with two copies, one containing the condition **colour of wine = white**, and one containing the condition **colour of wine = rosé**.

4.7 KB filtering

By this point, KRUST has generated a number of refined KBs. It now applies a set of filters to remove the least successful, based primarily on their performance on particular training examples. Currently, two KB filters are provided.

The refinement case filter. This rejects any KB which does not solve the refinement case correctly. (The procedure for generating refinements does not guarantee that refined KBs will necessarily behave correctly even for the original training case, because of rule interactions.)

The chestnut filter. The user is allowed to select priority examples, or "chestnuts", which the refined KBs must solve correctly. The chestnut filter rejects any KBs which do not solve all the priority examples correctly.

Priority examples were intended as a mechanism for representing important or typical cases within a domain. However, the priority example filter can also be used to assist with the iterative running of KRUST on a sequence of refinement examples (figure 4.6). The figure shows KRUST being run on a series of examples e_i. The initial faulty KB is KB_0. At each step, KRUST is given as input KB_{i-1} and refinement example e_i, and generates a best refined KB KB_i. This KB is then used as input to the next iteration. After each iteration, the refinement example e_i is added to the priority cases. The effect of this is to ensure that the chestnut filter rejects refinements which "undo" or otherwise interfere with refinements selected during earlier iterations.

4.8 Judgement

KRUST's final step identifies the most suitable from the remaining refined KBs. At this point, KRUST makes use of a set of further pre-classified examples, or *judging examples*, in

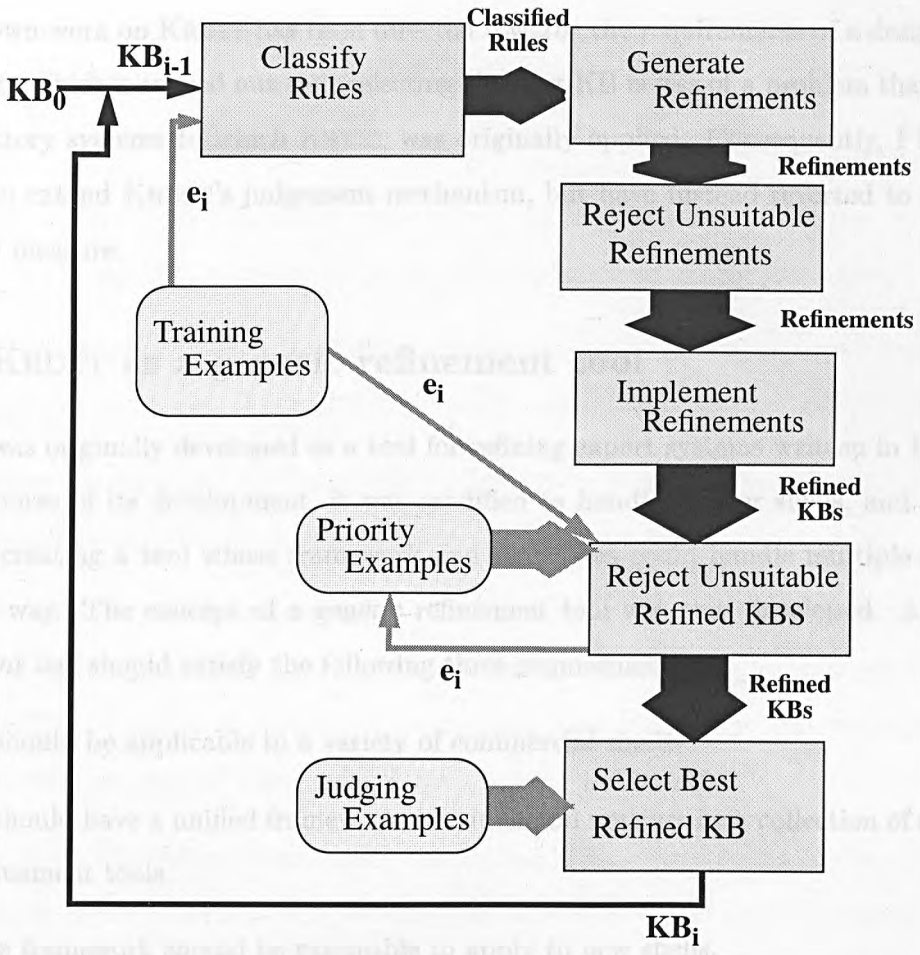

Figure 4.6: The iterative operation of KRUST

addition to the original refinement example, and the priority examples mentioned earlier. The initial set of generated refinements may be huge, but the number is reduced by the two filtering processes. Therefore, it is worthwhile using a fairly intensive empirical evaluation for the selection process. In addition, it turns out in practice that an evaluation based purely on the KBs' accuracy on the judging examples often fails to produce a unique best KB. Consequently, KRUST uses a SEEK-like procedure to assign blame to each rule in each refined KB, and then calculates the blame for each KB as the average blame for its rules. Craw (1991) reports that this metric may be used to distinguish KBs of equal accuracy, but is rarely in disagreement with the accuracy ranking. She also suggests that other KB metrics might be used to evaluate the remaining KBs instead of the blame metric described here, or alternatively, it may be convenient for the user to make the decision about which KB is the most suitable.

My own work on KRUST has been directed towards the requirements of a design application, for which it turned out that selecting the best KB is less of a problem than for the classificatory systems to which KRUST was originally applied. Consequently, I have not needed to extend KRUST's judgement mechanism, but have instead reverted to a simple accuracy measure.

4.9 KRUST as a generic refinement tool

KRUST was originally developed as a tool for refining expert systems written in PROLOG. In the course of its development, it was modified to handle further shells, and the goal arose of creating a tool whose framework and structures could handle multiple shells in a "tidy" way. The concept of a generic refinement tool was thus developed. A *generic refinement tool* should satisfy the following three requirements.

- It should be applicable to a variety of commercial shells.
- It should have a unified framework; i.e., it should not just be a collection of separate refinement tools.
- The framework should be extensible to apply to new shells.

The two principal features required in order to satisfy these requirements are:

- the ability to create an internal representation, or "knowledge skeleton", for each rulebase, using a common knowledge representation for rule elements; and
- extensible toolsets of filters and refinement operators.

It is clear that KRUST in its original form does not entirely satisfy these requirements, since it is applicable to only one shell. However, its use of an internal representation, and its use of tool-sets, facilitate its development into a generic refinement tool.

4.10 Conclusions

Craw et al. (1994) draw a distinction between Knowledge Base Refinement (KBR) and Theory Revision (TR), and shows that these two areas have different goals, and different strengths and weaknesses. The framework provided by this paper is useful for discussing

the strengths and weaknesses of KRUST, comparing it with other systems, and providing motivation for the work described in later chapters.

The principal distinctions between the KBR and TR systems are

Domains. KBR refines potentially noisy expert systems, whereas TR refines KBS which are more complex, possibly involving recursion, but which are noise-free.

Control Strategies. KBR refines expert systems using a variety of control and conflict resolution strategies, whereas TR is usually applied only to PROLOG programs.

Refinement operators. KBR generally includes more expressive operators than TR; for example, the ability to decrease or extend a numeric range.

Machine Learning Techniques used. KBR systems use techniques related to explanation-based learning (Boswell 1986), so require relatively few examples, but require access to either an oracle or additional background knowledge. TR systems tend to use induction, so require many examples.

Testing. Most TR systems require many examples; KBR systems need few examples, but require access to an oracle or background knowledge to perform refinement.

The principal conclusion of Craw et al. (1994) is that KBR systems are more compatible with actual KBSs than are TR systems.

It will be seen that according to this categorisation, KRUST is a KBR system. Its strengths and weaknesses are largely those of KBR systems in general. However, two features distinguish KRUST from both KBS and TR systems. Firstly, it generates many refinements initially and then selectively removes those refinements or refined knowledge bases that are shown to be ineffective. Filtering includes both the use of meta-knowledge and actual testing of the refined KBs. The generation and testing of large numbers of KBs increase the chance of KRUST's finding the correct refinement, allow it to detect unexpected interactions between refinements, and ensure that the refinements eventually returned give the correct results for the actual KBS. Secondly, it has a number of features listed in section 4.9 which facilitate its development into a generic refinement tool.

The strengths and weaknesses of KRUST may thus be summarised as follows.

Strengths

- Its ability to learn from a small number of examples is helpful for industrial applications, where classified examples are typically scarce.
- Its knowledge representation mechanism permits relatively small changes to individual rule conditions, such as adjustments to numeric thresholds, and adjustments to terms within a generalisation/specialisation hierarchy.
- It takes account of a KBS's control mechanism and conflict resolution strategy.
- Its hierarchical knowledge representation, with very general knowledge element types at the top of the hierarchy, makes it easy to extend the representation to include the knowledge elements from a variety of expert system shells.
- Its toolset of refinement operators is naturally extensible when new rule element types are introduced.
- Its ability to communicate directly with the shell interpreter gives KRUST access to the actual behaviour of the KB. This makes KRUST better able to take account of rule interactions than systems such as EITHER, which perform their filtering immediately after the refinement generation phase. An alternative approach would be to simulate the KB's behaviour, but this is less reliable. In addition, it would make it harder to extend KRUST to new shells, since a new simulator would have to be written for each.

Weaknesses

- Its lack of inductive operators prevents KRUST from learning new rule conditions, and greatly restricts its ability to learn new rules.
- KRUST relies on a single example for most of the refinement procedure, using other examples only for KB filtering and judging at a late stage. The use of multiple examples at an earlier stage might be expected to improve its learning ability.

The features of KRUST described above show that it is an appropriate choice for starting point for the development of a refinement tool which will be

- applicable to real KBSs as used in industry, and
- suitable for development into a generic tool, which is applicable to a variety of different KBS environments.

The following chapters show how my work has built on KRUST's strengths and addressed its weaknesses, in the following ways.

- KRUST's knowledge representation is extended to incorporate rule-elements from other KBSs, specifically PFES.
- The communication mechanism is adapted to cope with a situation where a KBS can not be queried directly.
- Inductive operators are introduced. These overcome two weaknesses in the present KRUST; they enable KRUST to learn rules and conditions which previously it could not have learned, and they allow it to make use of multiple training examples at an earlier stage of the refinement process.

Chapter 5

Adapting KRUST for use with PFES

This chapter describes the modifications and developments I have made to KRUST in order to apply it to the PFES shell. Many of the properties of PFES which required the adaptation of KRUST, such as the use of arithmetic expressions and a database of facts, are shared with other shells, so that the techniques I used to solve them are applicable outside this particular application. The following extensions to KRUST are described.

- A new mechanism was implemented for communicating between KRUST and PFES KBSs, since the method used for PROLOG KBSs was not applicable (section 5.1).
- A mechanism was introduced for deriving a refinement problem from PFES's complex output (section 5.2).
- The behaviour of forward-chaining rules was analysed, and it was determined that no modification was required to KRUST's refinement procedure (section 5.3).
- KRUST's knowledge representation was extended to take account of features of PFES not previously encountered (section 5.4).
- KRUST's refinement methods were modified to handle these new features of PFES (section 5.5).

A summary of the work described in this chapter has been published as Boswell, Craw & Rowe (1996).

5.1 Communication between KRUST and PFES

KRUST needs to communicate with PFES to obtain information about the KBS's behaviour. The original KRUST communicated with a KBS by submitting queries in the form of rule conclusions. It received two items of information in response: whether the conclusion was provable by the KB; and if it was provable, then what variable bindings resulted. However, PFES does not allow the submission of such queries; the only possible interaction is to submit the standard input and run the program to generate a formulation. Consequently, communication with PFES had to be handled in a different manner.

```
1. (GET-DRUG-DOSE 60 C
     (F-S-ATTRIBUTE-HAS-VALUE FORMULATION DRUG <DRUG> <>)
   ((<DRUG> . DRUG-A) (<> . WORLD-ROOT)) T)
2. (GET-DRUG-DOSE 60 C
      (F-S-ATTRIBUTE-HAS-VALUE SPECIFICATION TRY-NUMBER <NO> <>)
   ((<NO> . 1) (<DRUG> . DRUG-A)) T)
3. (GET-DRUG-DOSE 60 A (GET-USER-ANSWER DOSAGE-QUESTION NIL <DOSE> <>)
   ((<DOSE> . 10) (<NO> . 1) (<DRUG> . DRUG-A)) T)

4. (WEIGHT-TOO-LOW 77 C
     (F-S-ATTRIBUTE-HAS-VALUE SPECIFICATION
        MINIMUM-TABLET-WEIGHT <MIN-WEIGHT> <>)
      ((<MIN-WEIGHT> . 100) (<> . WORLD-ROOT)) T)
5. (WEIGHT-TOO-LOW 77 C
    (F-S-ATTRIBUTE-HAS-VALUE SPECIFICATION
        TARGET-TABLET-WEIGHT <WEIGHT> <>)
     ((<WEIGHT> . 100) (<MIN-WEIGHT> . 100) (<> . WORLD-ROOT)) T)
6. (WEIGHT-TOO-LOW 77 C (IS <WEIGHT> LESS-THAN <MIN-WEIGHT> <>)
     ((<WEIGHT> . 100) (<MIN-WEIGHT> . 100) (<> . WORLD-ROOT)) NIL)
```

Figure 5.1: A small part of a PFES trace file

The solution lay in the use of PFES's tracing mechanism. PFES is able to generate a trace file recording details of its activity, and it turns out that the information previously derived from queries can also be extracted from this file. Figure 5.1 shows a small selection from a trace file. (The numbers on the left were added later). Each entry in the trace describes a single attempt to satisfy a rule element. The entry records whether the attempt succeeded or failed, as well as the bindings of all variables which appear in the rule element. Entries contain the following items.

Rule name.

Index number. When PFES attempts to fire a rule, it tests each condition in turn, and each test generates an entry in the trace file. The series of tests associated with a single attempt to fire a rule are assigned an index number. For example, the first three entries in figure 5.1 represent a successful attempt to fire the rule GET-DRUG-DOSE, and they all have index number 60. This indicates that there have been 59 other rule firing attempts before the firing of GET-DRUG-DOSE.

Element type. 'C' for condition, 'A' for action.

Rule element. The condition or conclusion.

Bindings. The bindings of all variables occurring in the condition.

Result. T or NIL, standing for success or failure. Note that actions always succeed.

The first three records show the two conditions of rule GET-DRUG-DOSE succeeding, and the rule's action being executed. The next two records show the first two conditions of the rule WEIGHT-TOO-LOW succeeding. The last record shows the last condition of WEIGHT-TOO-LOW failing, because a WEIGHT of 100 is not less than a MIN-WEIGHT of 100.

This is exactly the type of information that KRUST requires about the behaviour of a KBS.

- Is rule R satisfied, and what variable bindings result?
- If rule R is not satisfied, which condition(s) cause it to fail?

PFES's trace file can provide answers to both these questions, with one small exception. When a rule fails to fire, the entries in the trace for that rule will include details of the first unsatisfied condition only, since later conditions will never have been tested. This potential difficulty, and its solution, is discussed in section 5.5.1.

The nature of the trace file means that KRUST can now determine more directly whether a rule has fired. When communicating with a PROLOG database, KRUST could determine whether a rule had fired only by testing each condition in turn. However,

when reading a PFES trace, KRUST can simply search for entries representing the action statement of the given rule. For example, the rule classification routine uses a predicate

```
(succeeded-with-bindings rule_name bindings)
```

which determines whether `rule_name` succeeded with the given bindings. If this predicate is called with the following arguments

```
(succeeded-with-bindings GET-DRUG-DOSE ((_DOSE . 10)))
```

then KRUST looks for entries in the trace file in which an action from rule `GET-DRUG-DOSE` has been executed with <DOSE> bound to 10. Entry 3 in figure 5.1 records such an action, so the function returns T.

A second KRUST function which returns information about KBS behaviour has the form

```
(not_fires rule_name bindings)
```

This returns the first condition to fail for each unsuccessful attempt to fire `rule_name`. It also returns the bindings for all variables in each failing condition. Thus, if it is called with the following arguments

```
(not_fires WEIGHT-TOO-LOW nil)
```

it will return the condition

```
is <WEIGHT> less-than <MIN-WEIGHT>
```

with variable bindings

```
(<WEIGHT> . 100) (<MIN-WEIGHT> . 100)
```

This information is used by KRUST to determine the condition in WEIGHT-TOO-LOW which should be generalised in order that the rule may fire, and also what degree of generalisation is necessary. For example, the rule WEIGHT-TOO-LOW could be enabled to fire by changing the comparison operator `less-than` to `less-than-or-equal`.

5.1.1 Communication between different platforms

KRUST runs on a Sun workstation and PFES on a PC, so some form of networked communication is required between the two programs. The data to be transmitted consists of a small number of files, each of which must be written by one of the programs and read by the other (problem input, trace, formulation and refined KBs). For this reason, I used PC-NFS to enable communication between the programs; it permits PFES to read files written by KRUST, and *vice versa*.

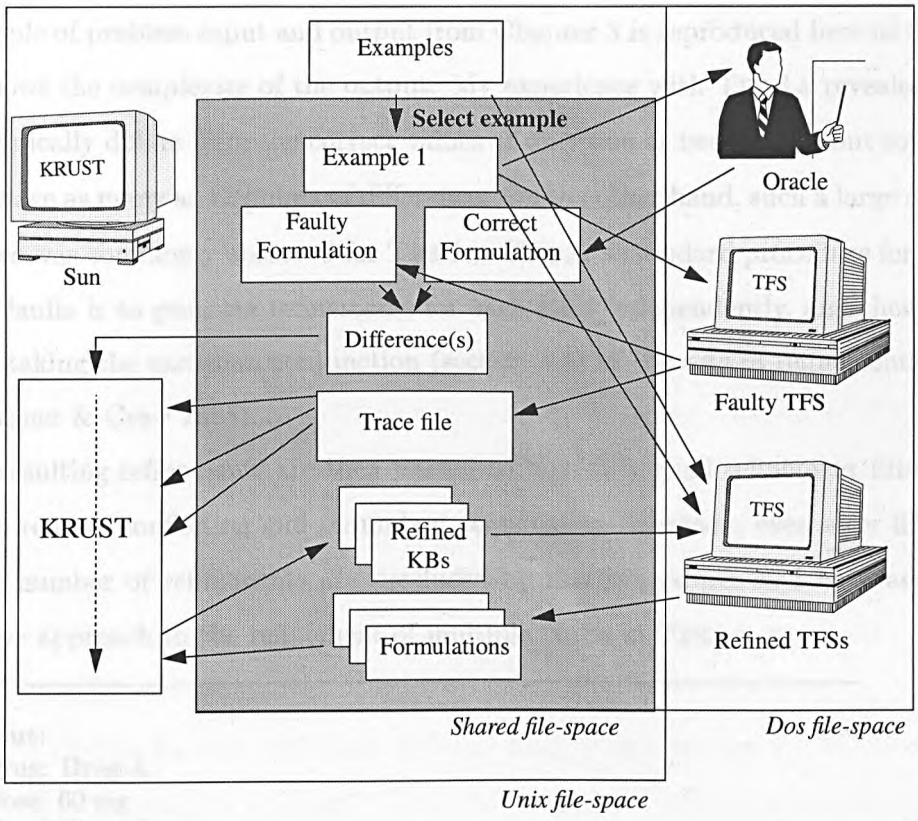

Figure 5.2: Interaction between Sun and PC during the running of KRUST

Figure 5.2 shows the interaction between Sun and PC for a single run of KRUST. First, an example is selected – Example 1 in the figure. This is then passed to the faulty version of TFS, and to an oracle. The two resulting formulations are compared, and the differences used to generate input to KRUST. In addition, the faulty TFS generates a trace file. While KRUST is running, it generates queries about the behaviour of TFS which are answered by searching through the trace file. Once KRUST has generated a set of refined KBs, it needs to test them on the refinement and judging examples, so it passes the KBs to the PC. The PC then runs these refined KBs on the examples, and passes the resulting formulations back to KRUST. Finally, KRUST compares these formulations with the oracle's formulations, and is thus able to select the best refined KB.

5.2 PFES's output - selecting the refinement problem

One consequence of PFES's formulation task is that its output is a compound answer, in contrast to the single result typically output from a diagnostic system. For convenience,

the example of problem input and output from Chapter 3 is reproduced here as figure 5.3, which shows the complexity of the output. My experience with TFS-1A revealed that its output typically differs from the correct values at only one or two points, but some examples can have as many as 12 points of difference. On the other hand, such a large number of differences was commonly observed for TFS-1B. KRUST's standard procedure for handling multiple faults is to generate refinements for each fault independently, and then combine them by taking the cartesian conjunction (section 4.5) of the sets of refinements for each fault (Palmer & Craw 1995).

The resulting refinements are then passed to KRUST's usual refinement filtering procedure to remove conflicting and redundant refinements. However, even after filtering an excessive number of refinements are produced by this procedure, so I have adopted an alternative approach to the refinement of multiple faults in TFS.

User Input:
Drug: Drug-A
Dose: 60 mg
No of fillers: 2

Specification:
full-stability: Yes
drug-filler–concentration: 0.9
minimum-tablet-weight: 100mg
maximum-tablet-weight: 800mg
target-tablet-weight: 260mg
start-strategy: strategy-A
filler-concentration: 66.9%
typical-disintegrant: Maize-starch
disintegrant-concentration: 0.05
tablet-weight: 252.2mg
total-concentration: 97%
tablet-diameter: 8.73mm
...various other properties...

Formulation:
Tablet weight: 250mg
Fillers: Lactose 66.7%,
Calcium phosphate 2.4%
Binder: Gelatin 4.1%
Lubricant: Stearic acid 1.0%
Disintegrant: Croscarmelose 2.1%

Figure 5.3: An example of user input, specification and formulation

This approach takes advantage of the fact that TFS calculates the values in the specification and formulation in a fixed order, and that later values are frequently dependent on earlier values. Consequently, if an early value is incorrect, it is likely that all the values

dependent on it will also be incorrect. Therefore there is no point in generating refinements for all the related faults. The sensible procedure, for an automated debugger as for a human one, is to fix the independent values first, and if other faults are still observed, deal with them in subsequent iterations.

The following procedure is used to determine the independent incorrect attribute(s). A *dependency chain* linking attributes a and z is defined to be a sequence of rules $R_1, R_2, \ldots, R_n$ where R_1 includes a condition referring to the value a, R_n includes a conclusion setting the value of z, and for each pair R_k, R_{k+1} the conclusion of R_k matches a condition of R_{k+1}. The existence of such a chain indicates that the value of a *may* affect the value of z, so we call z a *dependent* attribute. Given a set of faulty attributes A, for each $a_i \in A$ KRUST removes from A all attributes which depend on a_i. It then uses those that remain as input to the refinement process. For example, given the input

Dose 360 mg, Drug A, Number of fillers 2

the outputs of TFS-1A and the oracle differ in many respects (figure 5.4). However, all these attributes depend on TARGET-TABLET-WEIGHT, so that discrepancy for the refinement case submitted to KRUST has the following form.

System: TARGET-TABLET-WEIGHT of SPECIFICATION = 400
Oracle: TARGET-TABLET-WEIGHT of SPECIFICATION = 450

Incidentally, it may seem surprising that the system and oracle have different values for the number of fillers, since this attribute is included in the user input. The explanation is that the user chooses the *preferred* number of fillers; the formulator may override this choice if necessary, and on this occasion the oracle has done so.

5.3 Forward-chaining rules

This section discusses the effect of PFES's task-based control, and its rule chaining direction, on the rule classification and refinement process. First it shows that the process of refinement is independent of PFES's tasks, and to a large extent independent also of the rule chaining direction. It then describes the different ways in which forward-chaining rules from the same conflict set can interact, and shows how these different interactions affect refinement.

Specification Attribute	System value	Oracle value
Target-Tablet-Weight	400	450
Drug-Concentration	9/10	4/5
Filler-Concentration	0.0	0.1
Current-Strategy	Strategy-L	Strategy-A
Acceptable-Stability	(80 150)	(90 150)
Acceptable-Yield-Stress	(240 600)	(80 160)
Acceptable-Srs	(0 10)	(5 25)
No-Of-Fillers	2	1
Formulation Attribute	**System value**	**Oracle value**
Tablet-Weight	388.0	427.5
Tablet-Weight	390	430
Filler-List	(LACTOSE 0.0)	(CALCIUM-PHOSPHATE 0.105)
Binder	(GELATIN 0.041)	(GELATIN 0.021)

Figure 5.4: Differences in the output of TFS-1A and the oracle

5.3.1 The role of tasks in refinement

PFES is a task-based system, where each task has an associated rule-set. This section shows that rule chaining is independent of the division of rules into rule-sets, so that the refinement process need not take account of PFES's tasks.

A rule R_1 is said to chain with a rule R_2 if the conclusion of rule R_1 satisfies a condition of rule R_2. Rule chaining in PFES is accomplished by the setting and reading of attribute values, or by the writing and reading of agenda items. For example, a rule which adds an item to an agenda chains with a rule which reads an item from the same agenda. Since attribute values and agendas are independent of tasks, it follows that KRUST's analysis of rule-chaining can ignore the fact that rules are assigned to different tasks. For example, a rule in task CHOOSE-FILLER includes the action:

```
set the value of ACCEPTABLE-STABILITY to be <TARGET-STABILITY>
```

and another rule in task APPLY-FILLER-STRATEGY includes the corresponding condition:

```
ACCEPTABLE-STABILITY has value <MIN-STABILITY>
```

The allocation of these two rules to different tasks means that the second rule will not be able to fire until task APPLY-FILLER-STRATEGY has been invoked, which may happen before or after task CHOOSE-FILLER is completed. Nonetheless, once the second rule *is* given a chance to fire, it is able to read the value written previously by the first rule. Therefore, for purposes of refinement, the first rule may be regarded as chaining with the

second.

5.3.2 The refinement of forward-chaining rules

The rules within each of PFES's rule-sets are executed by forward-chaining. PFES is therefore a data-driven system. In contrast, the PROLOG KBSs to which KRUST was originally applied were backward-chaining, or goal-driven. However, since KRUST's analysis of a KBS's behaviour is goal-driven, KRUST works *backwards* from the conclusion to the initial facts regardless of the order in which the rules were originally executed. For example, suppose a rule base contains the fact A, rule 1: $A \rightarrow B$ and rule 2: $B \rightarrow C_{-}$, where C_{-} is an erroneous conclusion. Under forward-chaining, rule 1 will fire first, then rule 2. However, the trigger for refinement is the conclusion C_{-}, since that is the only aspect of the system's behaviour initially known to be wrong. The blame allocation process will then work *backwards* from C_{-} to rule 2 and then rule 1.

However, although refinement can proceed backwards from the goal, regardless of the firing direction of the rules, there are situations in which the behaviour of forward and backward chaining rules differ. When dealing with backward-chaining rules, KRUST makes the assumption that inference will stop when an end-rule fires, so that adjusting the relative priority of end-rules is a useful operation. However, no such cessation applies to other rules, which will fire as required to achieve any given goal, so changing *their* priority serves no purpose. The rule behaviour for forward chaining rules is slightly different. To examine this, it is useful to introduce the concepts of *potentially clashing rules* and *self-disabling rules.*

5.3.3 Potentially clashing rules

Potentially clashing rules are those for which there exists a rule condition which matches each of their conclusions. For example, consider the following three rules.

```
Rule INSOLUBLE-FILLER-RULE
If
    ...
Then
    set the value of BINDER in FORMULATION to be GELATIN
```

```
Rule INSOLUBLE-DRUG-NOT-FILLER
If
     ...
Then
     set the value of BINDER in FORMULATION to be PVP

Rule SOLUBLE-INGREDIENTS-RULE
If
     ...
Then
     set the value of BINDER in FORMULATION to be MAIZE-STARCH
```

The conclusions of these rules all match the condition

```
BINDER has value <BINDER> in FORMULATION
```

so the rules are potentially clashing.

In principle, the behaviour of potentially clashing rules under forward-chaining can be of four types, depending on the rule-element types which form their conclusions.

Overwriting. This describes rules whose conclusions write to a slot which can hold only one value, so that each rule in turn overwrites the value written by the previous rule. This is possible in PFES, but does not occur in TFS.

Successive. This describes rules whose conclusions are effectively writing to a multiple-value slot, so that each rule causes values to be appended to the values contributed by earlier rules. Examples of this type are rules that conclude ADD ... TO BOTTOM OF AGENDA in TFS.

Reverse successive. This category is similar to 'successive', except that the values being written are stored in a stack rather than a queue, so that the value written last is read first. For example, the conclusion ADD ... TO TOP OF AGENDA in PFES would behave in this way, but does not occur in TFS.

5.3.4 Self-disabling rules

A self-disabling rule is defined as a rule whose conclusion matches one of its exceptions. These occur frequently in TFS, for example.

```
Rule INORGANIC-FILLER-RULE
If
    <STABILITY> is after GELATIN on STABILITY-AGENDA
    <STABILITY> is greater-equal 90
    ...
Then
    set the value of BINDER in FORMULATION to be GELATIN
Unless
    BINDER has value <ANY> in FORMULATION
```

Here the conclusion sets the value of binder, and the exception checks whether the value of binder has already been set. INORGANIC-FILLER-RULE is one of a set of self-disabling rules for choosing a binder. When any one of them fires and sets a value for the binder, all the other rules are disabled.

Consider next the behaviour of a group of potentially-clashing rules which are also self-disabling. As soon as one of the rules fires, all the others will be disabled. Consequently, only the highest priority rule in the group that is satisfied will fire. Note that this behaviour is similar that to a group of potentially-clashing end-rules under backward-chaining, where again only the highest-priority rule will fire.

5.3.5 The refinement of potentially-clashing rules

The effect of rule priority on the behaviour of potentially-clashing forward-chaining rules depends on the type of their conclusions. For rules with successive conclusions, as defined in section 5.3.3, and for self-disabling rules, high priority rules take precedence, so KRUST's original algorithm is still appropriate. For overwriting and reverse successive rules, low priority rules would take precedence, though this phenomenon does not in fact occur in TFS, so KRUST has not yet been adapted to handle such rules.

5.4 Rule element representation

One goal in the development of KRUST is the creation of a generic refinement tool, applicable to a variety of KBSs. When KRUST is applied to a KBS, it has to build an internal

representation of the rules in the KBS, in order to reason about the way in which the rules may interact. KRUST therefore requires a generic knowledge representation schema, within which any rule from a currently refinable KBS can be represented. This section describes KRUST's knowledge representation schema.

5.4.1 Terminology

First it is necessary to introduce some terminology. It is assumed that rules to which KRUST is applied will have the following form.

If Condition$_1$
Condition$_2$
...
Condition$_n$
Then Conclusion$_1$
Conclusion$_2$
...
Conclusion$_m$

Each condition and conclusion is said to be a *rule element*, and is made up of one or more *knowledge elements*. The reason why two different terms are required is that not all knowledge elements are rule elements in their own right. A rule element is therefore a special class of knowledge element; one that can form a rule condition or conclusion. For

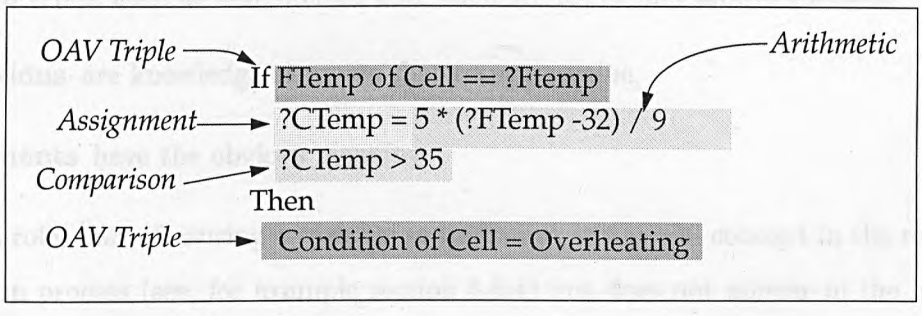

Figure 5.5: A rule, broken down into knowledge-elements

example, figure 5.5 shows a rule broken down into knowledge elements. (The meaning of the various knowledge element types mentioned in figure 5.5 is discussed below). The rule's second condition is the assignment `?CTemp = 5 * (?Ftemp - 32) / 9`. This is itself a

knowledge element, and contains within it a smaller knowledge element, the arithmetic expression `5 * (?Ftemp - 32) / 9`. The assignment forms a rule condition in its own right, so is a rule element. However, it would be meaningless to include an isolated arithmetic expression as a rule condition or conclusion, so arithmetic expressions are *not* rule elements.

5.4.2 The knowledge element hierarchy

The goal of KRUST's knowledge representation is to classify the knowledge elements which occur in KBS rules, and to arrange them in a meaningful hierarchy.

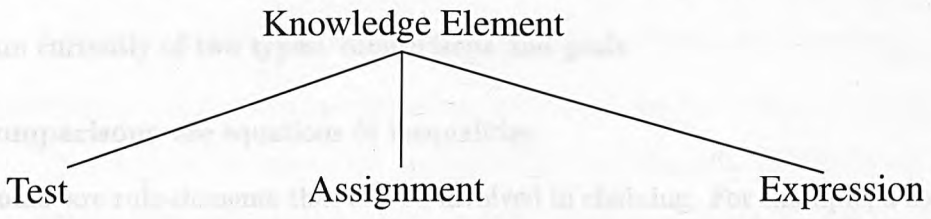

Figure 5.6: KRUST's basic knowledge elements

Although it first appears that there is a wide variety of representations used by these tools, there are in fact only a limited number of roles that a knowledge element can play. Three basic classes of rule elements have been identified, corresponding to these roles (figure 5.6).

Tests are conditions that can succeed or fail, for example, comparisons. All other condition types, such as assignments and calculations, should always succeed.

Expressions are knowledge elements that return a value.

Assignments have the obvious meaning.

A fourth role, that of causing a variable to be bound, is a useful concept in the refinement generation process (see, for example section 5.5.4) but does not appear in the hierarchy, since it is a property of the way a rule element is used, rather than of the rule element itself.

These roles can be further sub-divided, giving the hierarchy shown in figure 5.7. It will be seen that tests and expressions both have a number of sub-types; these are described below.

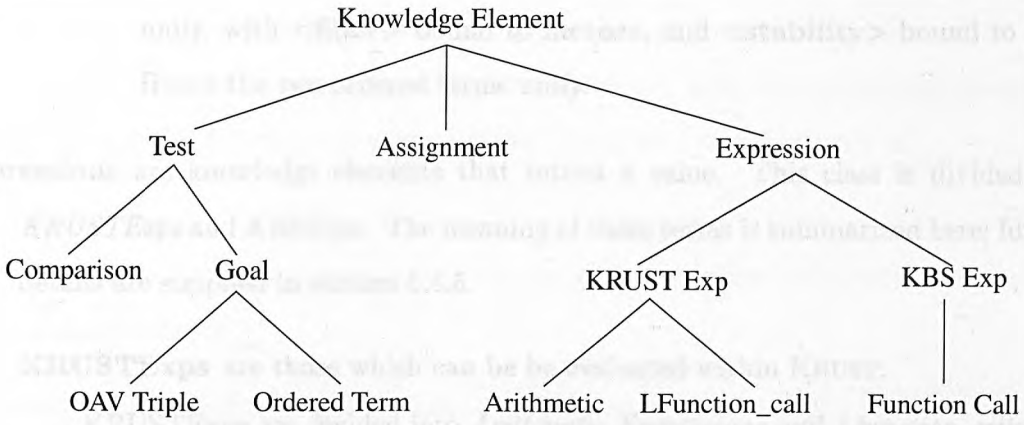

Figure 5.7: KRUST's hierarchy of knowledge elements

Tests are currently of two types: *comparisons* and *goals.*

Comparisons are equations or inequalities.

Goals are rule elements that can be involved in chaining. For example, a conclusion that sets a value can chain with a condition that uses that value, so both are classified as goals. There are two types of goal: OAV triples, and *ordered_terms.*

OAV triples have been simplified since the original KRUST implementation; they no longer include upper and lower bounds. Instead, such bounds are now represented by separate comparison conditions. For example, the condition that the stability of filler is greater than 90 could previously have been represented by a single upper-bounded OAV:

stability of filler > 90

KRUST now represents this condition as an OAV followed by a comparison:

stability of filler is <stability>

<stability> < 90

Ordered terms consist of a keyword followed by arguments, as in CLIPS. Two ordered terms unify if they have the same keyword and arity, and the corresponding arguments unify. For example, consider these two ordered terms.

agenda-attribute(property-agenda, <filler>, <stability>)

agenda-attribute(property-agenda, lactose, 96.9)

They both have arity 3, keyword **agenda-attribute**, and their arguments

unify, with <**filler**> bound to **lactose**, and <**stability**> bound to 96.9. Hence the two ordered terms unify.

Expressions are knowledge elements that return a value. This class is divided into *KRUSTExps* and *KBSExps*. The meaning of these terms is summarized here; further details are supplied in section 5.4.5.

KRUSTExps are those which can be be evaluated within KRUST.

KRUSTExps are divided into *Arithmetic Expressions* and *Lfunction_calls*.

Arithmetic Expressions make use of the four operators $+, -, \times, /$.

Lfunction_calls are expressions which can be translated into Lisp.

KBSExps include all those expressions which can not be evaluated within KRUST. They are passed back to the original KBS for evaluation. KBSExps are intended to deal with the situation where an expert system shell allows calls to procedural code, such as C functions, in rule elements.

All the knowledge elements appearing in figure 5.7 have now been defined. An example of a rule, broken down into knowledge elements, appears here for convenience as figure 5.8, which is a copy of figure 5.5. There are two aspects of the hierarchy which require further explanation: the degree to which rule elements are actually nested, and the use of KRUSTExps and KBSExps by TFS.

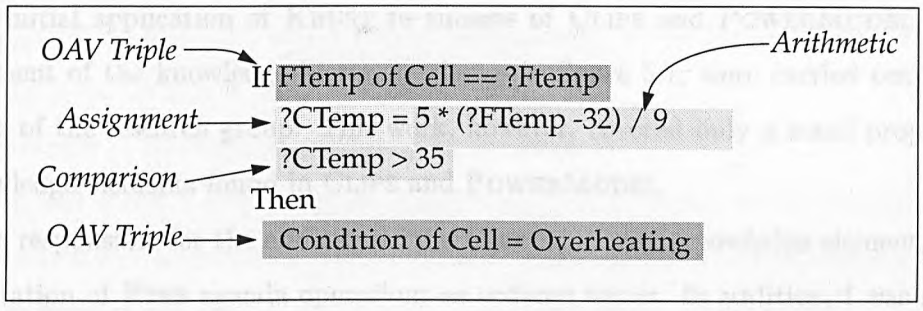

Figure 5.8: A rule, broken down into knowledge-elements

5.4.3 Nested knowledge elements

In principle, each rule element can consist of an arbitrarily deeply nested structure of knowledge elements. In practice, most rule elements consist of a single knowledge element,

with no nesting. The exceptions are assignments and arithmetic expressions. The right-hand side of an assignment can itself be an expression, and arithmetic expressions are defined recursively. Consider for example, the following TFS rule condition

```
(lisp-to-value
   (* 100 (/ (- <TABLET-YP-FAST> <TABLET-YP-SLOW>) <TABLET-YP-SLOW>))
   <TABLET-SRS>)
```

The effect of this condition is to perform the calculation

```
100 * (<TABLET-YP-FAST> - <TABLET-YP-SLOW>) / <TABLET-YP-SLOW>
```

and then to store the result in the variable <TABLET-SRC>. It is therefore translated into the following KRUST assignment, where the right-hand side of the assignment is itself an arithmetic expression.

```
_TABLET-SRC = 100 * (_TABLET-YP-FAST - _TABLET-YP-SLOW)
                  / _TABLET-YP-SLOW
```

5.4.4 Authorship of the knowledge hierarchy

The only knowledge element used by the original KRUST was the OAV triple. Since then, other knowledge elements were introduced to serve the needs of other KBS development tools: CLIPS, POWERMODEL and PFES.

The initial application of KRUST to subsets of CLIPS and POWERMODEL, and the development of the knowledge hierarchy shown in figure 5.7, were carried out by other members of the research group. This work, however, covered only a small proportion of the knowledge elements found in CLIPS and POWERMODEL.

I was responsible for the addition of the **Lfunction_call** knowledge element, and the representation of PFES agenda operations as ordered terms. In addition, I was responsible for the more recent extensions of the knowledge hierarchy described in section 5.4.6. These permit the representation of more complex CLIPS rule elements, which could not be represented within the hierarchy shown in figure 5.7.

5.4.5 The use of KRUSTExps and KBSExps by TFS

The fact that PFES is unable to accept individual queries means that KRUST can not pass KBSExps to PFES for evaluation, so all value-returning expressions must be represented as KRUSTExps. However, this is not a problem, given that KRUST is written in Lisp and PFES rules use a Lisp-like syntax; in particular all value-returning expressions in PFES rules are written in Lisp.

If PFES had not been written in Lisp, then two approaches to handling PFES expressions might have been considered.

- Extend the PFES interface so that expressions *can* be passed to the KBS for evaluation. This would have required assistance from the PFES developers.
- Provide KRUST with an interpreter for PFES expressions.

PFES's use of Lisp facilitated KRUST's handling of PFES expressions, but one small difficulty remained. This was caused by TFS's use of a few simple functions which are built-in in PFES but not Lisp. There were two possible solutions to this; to translate the PFES functions into Lisp, or to provide KRUST with definitions of these functions. I could find no compelling reason to prefer one approach over the other, and in fact adopted the second. An example of such a TFS function is **min-val**, which takes as argument a list of numbers and returns the minimum. **min-val** is used for example in the TFS condition

```
is <MIN-SRS> equal-to (min-val <RANGE>)
```

This condition is translated into a KRUST comparison, where the right-hand side of the comparison is an Lfunction_call.

<min-srs> = lfunction_call(min-val, <range>)

Since **min-val** is *not* a Lisp function, KRUST must be provided with a definition of this function which can be executed when required. **min-val** can easily be defined using the Lisp function **min**.

5.4.6 Further development of the knowledge hierarchy

The hierarchy shown in figure 5.7 has grown during the course of the development of KRUST, and it is expected that further terms will be added in the future as new KBS shells require. However, the use of a hierarchy allows new rule elements to be added

within a consistent framework, and encourages the implementation of procedures which take advantages of properties shared between different elements. This can be illustrated with reference to recent work which I have carried out as part of the KRUSTWorks[1] project, which aims to develop KRUST further as a generic tool. This work has required KRUST to be applied to the CLIPS and POWERMODEL shells. These include a number of novel rule elements which were *not* dealt with in the course of the initial work on these applications by other members of the KRUST group. Two such elements from CLIPS, whose representation within KRUST I have recently implemented, are unordered facts, and field constraints within patterns.

CLIPS unordered facts are represented by a template name, followed by a series of keyword/value pairs, for example

```
(excipient (name mannitol)
           (yp 90.2)
           (solubility 166))
```

Since unordered facts can chain, but do so in a slightly different way from OAVs and ordered_terms, they are represented within the knowledge hierarchy by a new sub-class of goal called **av_tuple**. In addition, a new method has been written to define chaining for av_tuples.

The second novel type of rule element, field constraints, appear as tests within CLIPS patterns. For example, the following pattern

```
(temperature water ?t & : (> ?t 60))
```

matches facts of the form

```
(temperature water n)
```

where n > 60. This pattern can be represented in KRUST by a comparison whose left-hand side is an OAV, as follows:

```
(temperature of water is ?t) > 60
```

This did not require any extension of the knowledge hierarchy, but it did require modifications to those operators that apply to comparisons.

[1] KRUSTWorks: Developing a Toolkit for Automated Knowledge Refinement. EPSRC grant GR/L38387

Thus, experience with PFES, CLIPS and POWERMODEL has shown that KRUST's knowledge hierarchy is readily extensible to handle the wide variety of rule elements found in different shells.

5.4.7 The PFES database

The knowledge in PROLOG KBSs that can be refined by KRUST is divided into two parts: the KB, consisting of a set of rules, and a set of facts associated with each example. The situation with PFES is more complex, since the knowledge is stored in *three* parts: rules, the facts associated with each example (the drug, dosage and number of fillers), and a database of chemical facts, or background knowledge. The rules and facts correspond to those in the PROLOG KBSs, but the knowledge in the database is new.

As an example of the records appearing in the database, here is the entry for the filler MANNITOL.

```
(DEFOBJECT MANNITOL
   (YP 90.2)
   (YP-FAST 161.0)
   (SRS 78.5)
   (SOLUBILITY 166)
   (IS-A FILLER)
   MIXES-WITH (MAGNESIUM-CARBONATE CALCIUM-PHOSPHATE
               CALCIUM-DIHYDROGEN-PHOSPHATE))
```

In order to represent these facts within KRUST's existing framework, they are translated into rules with an empty condition list. For example, the first few facts in the above database entry are translated into the following KRUST rules.

```
RULE MANNITOL_RULE1
If
Then
    YP of MANNITOL = 90.2

RULE MANNITOL_RULE2
If
```

```
Then
    YP-FAST of MANNITOL = 161.0

RULE MANNITOL_RULE3
If
Then
    SRS in MANNITOL = 78.5
```

The presence of these conditionless rules, derived from database entries, permits the creation of a new refinement implementation operator designed to deal with the situation where a fact is missing from the database. This is described in section 5.5.1.

5.4.8 Agendas

There is a group of rule elements that appear at first to be unique to PFES, and therefore potentially difficult to represent within a common framework. These *agendas*[2] are untyped lists, where items can be written and read from the top or bottom of the list. Additionally, items can be read from directly below any other item on the list. Agendas are used to pass data between rules which generate values and those which subsequently test or filter them. For example, the following rule copies fillers from STABLE-FILLER-AGENDA to FILLER-AGENDA.

```
Rule GET-STABLE-FILLERS
If
    <FILLER> is on STABLE-FILLER-AGENDA
Then
    add <FILLER> to bottom of FILLER-AGENDA
```

The following rule then removes fillers whose concentration is too high, thus functioning as a filter.

```
Rule REMOVE-EXCESSIVE-FILLERS
If
    <FILLER> is on FILLER-AGENDA
```

[2]PFES agendas have no connection with the standard scheduling connotation.

```
MAX-LEVEL has value <LEVEL> in <FILLER>
FILLER-CONCENTRATION has value <CONC> in SPECIFICATION
<CONC> is-greater-than <LEVEL>
```

Then

```
remove <FILLER> from FILLER-AGENDA
```

Agendas can thus be seen as a mechanism for storing data, usually data that has passed some test. This suggests that the procedural commands for adding items to an agenda, and then reading them off it, might be translated into declarative statements making use of attributes and values. For example, both the statements

```
<FILLER> is on AGENDA
```

and

```
add <FILLER> to bottom of AGENDA
```

could be represented by an OAV triple such as

```
AGENDA of <FILLER> is true
```

Whatever representation for agenda operations is used must chain in the same way the operations themselves do. The OAV representation satisfies this requirement. For example the PFES action

```
add <FILLER> to bottom of AGENDA
```

chains with the PFES condition

```
<FILLER> is on AGENDA
```

and the KRUST OAV representations (now the same for both action and conclusion) chain in the same way.

In practice, however, it turns out that there is a need to record more information than this, so rather than an OAV, an ordered term must be used. These too can be selected so as to chain in an appropriate way. The need for more information arises from the fact that other agendas have a more complex structure, so that a single agenda can store, for example, both excipients and their stabilities. The number of different structures actually employed within TFS is fairly limited. Two of the most common types are shown in figure 5.9. Each example shows the contents of an agenda at some point during the running of TFS, together with the PFES rule elements that write to and read from the agenda, and the KRUST representation of these elements.

Snapshot of FILLER-AGENDA

CELLULOSE LACTOSE CALCIUM-PHOSPHATE

PFES Read/Write Operations on FILLER-AGENDA

Conclusion: **add** <FILLER> **to bottom of** FILLER-AGENDA
Condition: <FILLER> **is on** FILLER-AGENDA

KRUST representation of each operation

on-agenda(FILLER-AGENDA, <FILLER>)

Snapshot of PROPERTY-AGENDA

CELLULOSE 46.2 LACTOSE 158.8 CALCIUM-CARBONATE 851.1

PFES Read/Write Operations on PROPERTY AGENDA

Conclusions: **add** <FILLER> **to bottom of** PROPERTY-AGENDA
add <STABILITY> **to bottom of** PROPERTY-AGENDA
Conditions: <FILLER> **is on** PROPERTY-AGENDA
<STABILITY> **is after** <FILLER> **on** PROPERTY-AGENDA

KRUST representation of each of these paired operations

on-agenda(PROPERTY-AGENDA, <FILLER>)
agenda-unlabelled-attribute(PROPERTY-AGENDA, <FILLER>, <STABILITY>)

Figure 5.9: Agendas and their PFES operations

The Filler-Agenda is simply a list of excipients; their presence on the agenda indicates that they have passed a stability test.

The Property-Agenda again shows a list of excipients, but now each excipient has an associated floating-point number, representing the value of a mechanical property.

In each of these cases, the rule elements which read and write the agenda items can be represented in KRUST as ordered terms, as shown in figure 5.9. However, the translation process is complicated by the fact that the semantics of the PFES conditions have sometimes to be determined by their context, whereas in the KRUST representation the semantics have to be explicit in each individual rule element. This requirement is imposed by the needs of the refinement procedures, which work with the KRUST internal representation. For example, the fact that <STABILITY> is a property of <FILLER> is implicit in the PFES statements which write a filler to the agenda, followed by its stability. The translator has to recognise that when two successive actions occur in the conclusion of a

PFES rule, one writing an excipient and the other a number, then the number is to be regarded as an attribute of the excipient. This semantics can then be made explicit in the KRUST representation, where the **agenda-unlabelled-attribute** statement includes both the filler and its stability as arguments. One consequence of this is that PFES commands of the type **add** <ITEM> **to bottom of** <AGENDA> have different KRUST representations, depending on whether <ITEM> represents an excipient or an attribute.

The patterns currently recognised by the translator are listed below. All the patterns are ways of writing a series of excipients to the agenda, where each excipient is optionally followed by a series of numbers representing values of properties of that excipient.

- The rule contains a single action to write a value to an agenda, so that the elements on the agenda will all be of the same type. This corresponds to the situation in FILLER-AGENDA.

- The rule contains a pair of successive actions, both writing values, where the second value is numeric. The fact that the value is numeric is identified by the fact that the PFES variable representing it includes the strings CONC or WEIGHT.

- The rule contains three successive actions, where the second and third both write numeric values. This is an extension of the previous case; this time, both the second and third items written are values of properties of the first item.

- The rule contains a sequence of actions, where the second and succeeding even numbered actions write a fixed string, rather than a variable, to the agenda. These strings may be regarded as labels for the values that follow.

Corresponding patterns may be observed in the conditions of rules that read from the agendas. In each case, both reading and writing actions may be translated into KRUST ordered terms, in a similar way to the examples shown in figure 5.9.

It will be seen that these patterns are application-specific and require the use of a little application-specific meta-knowledge to recognise; this is unfortunate, but the difficulty is caused by the flexible nature of the agenda structure, which makes it difficult to predict in advance how it is going to be used. A human seeking to understand the code would have the benefit of comments and meaningful variable name; an automatic translator requires background knowledge and examples of programming clichés.

5.4.9 Rule translation

During the extension of KRUST to apply to CLIPS and POWERMODEL, another member of the research team developed a tool to assist with the development of rule translators (Palmer 1995). The user must provide grammar rules describing the syntax of a given KBS's rules, together with functions which will create the appropriate KRUST representation for individual knowledge elements. The tool then uses these grammar rules to build a translator for the KBS. Other members of the team have built translators for a subset of CLIPS, PROLOG and POWERMODEL rules using this tool.

However, the tool could not be applied to PFES rules, because the elements in a PFES rule object can appear in an arbitrary order. Instead, I was obliged to write a separate translator by hand. However, the work involved in writing a PFES-specific translator was not much greater than would have been required to write grammar rules for use by the parser.

5.5 Extension of refinement procedures to handle PFES rule elements

PFES includes a number of features which were not possessed by the PROLOG KBS's to which KRUST was originally applied. This section describes changes and additions which I have made to KRUST's refinement procedures in order to handle these features. The features, and the corresponding extensions to KRUST, are as follows.

- PFES's use of a fact database requires the creation of an **add_fact** operator. When a gap in the database is identified, this operator creates the required new fact.
- The use of variables in PFES's rules requires the addition of a new class of target-rule: wrong-fire. I describe how wrong-fire rules are refined, and give an example. This example also illustrates how the refinement process is able to reason about arithmetic expressions.

5.5.1 Refining the fact database

The presence of a database of facts introduces two new faults which can arise in a KB: a missing fact, and an incorrect fact. This section describes the refinement operator

add_fact which can learn a missing fact. The mechanism for dealing with incorrect facts is described in section 5.5.3, followed by an example in section 5.5.4.

PFES facts are represented within KRUST as rules having no conditions, where the conclusion is an OAV. For example,

```
RULE LACTOSE_RULE4
If
Then
    SRS of LACTOSE = 21.3
```

Facts are read by rule conditions which chain with the conclusions of these rules, and bind a variable as a result. For example, the condition

```
SRS of LACTOSE = <SRS>
```

would chain with the above LACTOSE_RULE, binding <SRS> to 21.3.

The immediate effect of a missing fact will be that the rule condition which should chain with it will fail. If this prevents a rule from firing which ought to have fired, the result will be incorrect output from the KBS. Suppose a rule R fails to fire because a missing fact prevented condition C from being satisfied. Then if R is an end-rule, this will lead directly to incorrect output from the KBS, with R itself being classified as a no-fire rule. Alternatively, the failure of R may lead to the failure of a whole chain of rules, where the last rule in the chain becomes a no-fire rule. In either case, the result of R's failure is the appearance of a no-fire rule.

When KRUST tries to fix this no-fire rule, it will generate refinements to enable R, which it can do either by generalising C, or by enabling a rule whose conclusion matches C. The new **add_fact** operator is therefore associated with 'generalise'. Given a failed condition C in rule R, the **add_fact** operator will create a new fact which will match C. Moreover, it will choose the new fact in such a way as to satisfy subsequent conditions in R.

To limit the use of the **add_fact** operator, the user must provide it with a list of the attributes which are included in the database (a new form of meta-knowledge); the operator will generate a new fact only if the missing attribute appears in this list.

The remainder of this section uses an example to explain how the **add_fact** operator creates a fact which enables a previously unsatisfied rule. An example of a fault requiring

the application of this operator is shown in figure 5.10. This shows the rules responsible for a fault in which TFS incorrectly recommends the filler Calcium Phosphate. The reason is that the MAX-LEVEL of Calcium Phosphate is missing from the database, so that the second rule, REMOVE-EXCESSIVE-FILLERS, fails to fire and hence fails to remove Calcium Phosphate from the filler agenda. Hence Calcium Phosphate remains on the agenda to be read by GET-INSOLUBLE-FILLER.

An experiment is therefore created to generalise the failed condition

`MAX-LEVEL has value <LEVEL> in FILLER`

and since MAX-LEVEL appears in the list of database attributes, the **add_fact** operator can be applied. The purpose of **add_fact** is to choose a value that will enable the failing rule to succeed, so it has to consider the constraints imposed on this value by subsequent conditions. The operator performs the following actions.

1. Identify the variable which stores the value when it is read from the database. Here this is <LEVEL>.

2. Identify all comparison tests which apply to this variable. Here there is only one such test: `<CONC> is greater-than <LEVEL>`.

3. For each such comparison, locate all other variables appearing in the comparison, and identify the rule condition where they were bound. Here, the only other variable is <CONC>, which was bound in the condition `FILLER-CONCENTRATION has value <CONC>`.

4. For each of these other variables, determine their values. For variables that were bound before the failing condition was reached, this can be done by looking at the trace for the current rule, REMOVE-EXCESSIVE-FILLERS. For variables that were bound later, such as <CONC>, the trace for the current rule is no use, since execution stopped at the failing condition. Instead, KRUST attempts to determine a binding by looking at the trace for the rule(s) whose conclusion matches the binding condition. E.g., to determine the value of <CONC>, KRUST looks for rules which set the value of FILLER-CONCENTRATION. It then determines from the trace entries for these rules the value to which FILLER-CONCENTRATION was set. If the trace indicates that no such rule fired, it follows that the binding condition would have failed, so no binding

value can be deduced.

In general, such constraints will not be sufficient to determine a single value for the new fact, so an additional constraint is imposed: that the change caused by adding the new fact be the most conservative. In this case, the most conservative value for MAX-LEVEL is the largest, since the larger the value, the closer the KB's behaviour will be to that where the value is missing, viz., that the excipient is never removed.

The addition of new facts is an appropriate area for the application of induction to impose further constraints on the new fact. An inductive **add_fact** operator is described in section 6.2.

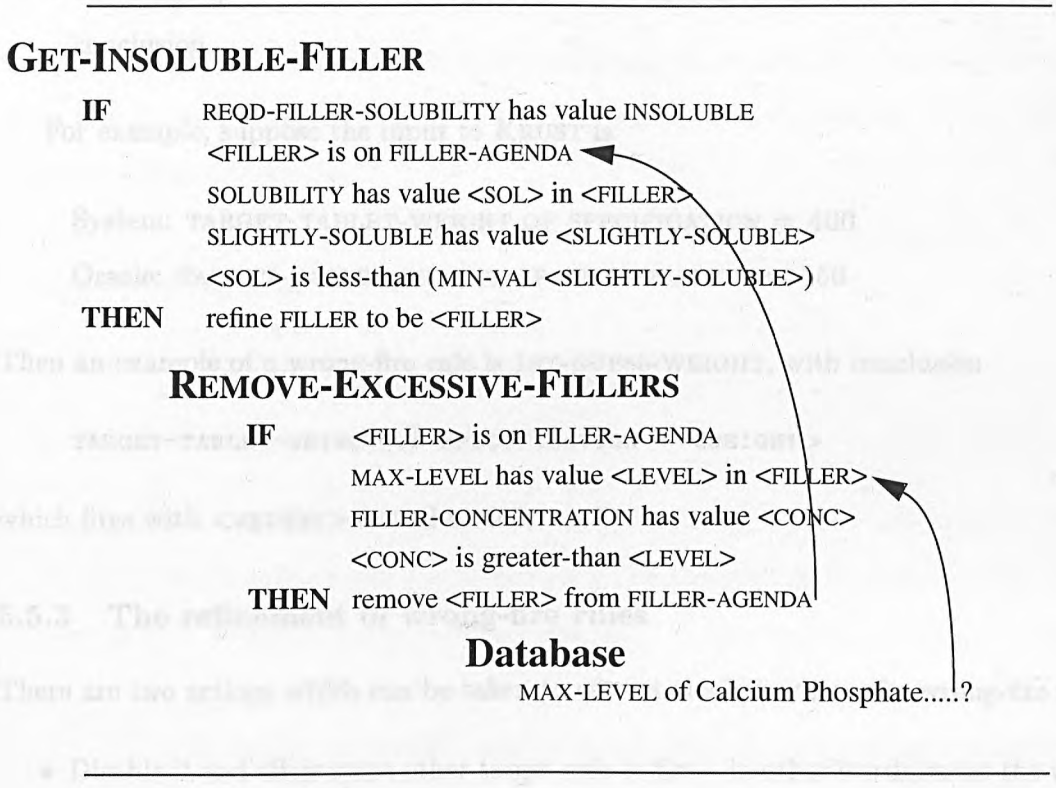

Figure 5.10: Rule chain for wrong filler example

5.5.2 Variables

This section explains how the use of variables in a KBS requires the introduction of a new class of target rule: wrong-fire rules. There follows an account of how KRUST refines wrong-fire rules, which is then illustrated by an example from TFS.

For propositional KBSs, KRUST identifies two basic rule classes: *error-causing* – rules

which fired and gave the wrong conclusion, and *no-fire* – rules which gave the right conclusion but did not fire. In order to apply KRUST to PFES, which uses predicates including variables as arguments, I have extended these by adding a new rule class, *wrong-fire*, to describe rules which fired and gave the wrong conclusion, but could have given the right conclusion if they had fired with a different set of variable bindings.

A wrong-fire rule is therefore identified during rule classification by the following features:

- its conclusion matches both the expert and the system conclusion, and
- the bindings with which it fired match the system conclusion but not the expert conclusion.

For example, suppose the input to KRUST is

System: TARGET-TABLET-WEIGHT OF SPECIFICATION = 400

Oracle: TARGET-TABLET-WEIGHT OF SPECIFICATION = 450

Then an example of a wrong-fire rule is 1ST-GUESS-WEIGHT, with conclusion

TARGET-TABLET-WEIGHT of SPECIFICATION = <WEIGHT>

which fires with <WEIGHT> bound to 400.

5.5.3 The refinement of wrong-fire rules

There are two actions which can be taken to correct the behaviour of a wrong-fire rule.

- Disable it and allow some other target rule to fire – in other words, treat the wrong-fire rule exactly as if it were error-causing, or
- Make some change to the KB which allows the wrong-fire rule to fire with the correct bindings.

To change the KB to allow the wrong-fire rule to fire correctly, KRUST does two things: it identifies the rule-element at which the incorrect binding is originally set, and then changes the rule element appropriately. There may be more than one such rule-element, in which

case a refinement is generated for each. The procedure for identifying the element where the value is set is described below, first informally, then more precisely.

Informally, KRUST follows the reasoning chain backwards to a point at which a value is hard-wired into a fact, or occasionally a rule conclusion. It then changes the fact or conclusion as required. This procedure is complicated when KRUST encounters an arithmetic expression. When this happens, KRUST generates a number of experiments:

- to change any one of the inputs to the arithmetic expression to give the desired result;
- to change the arithmetic expression itself to give the desired result.

More formally, the algorithm is as follows. Suppose the incorrect binding for variable V in the wrong-fire rule R is S (System) and the correct binding for V is O (Oracle). Then we define a procedure

---

`find-binding-condition(`R`, `V`, `S`, `O`).`

1. Let C be the condition in rule R where V is bound.
2. The next step depends on the type of C.

   Goal, with value uninstantiated. Let R' be the rule whose conclusion C' chained with C and bound V to S, and let V' be the term in C' which unifies with V. If V' is bound, terminate and return C' as the binding condition, with O as the corrected value. Here C' is likely to be the conclusion of a fact rule, but it may not be. Otherwise, apply the procedure `find-binding-condition(`R'`, `V'`, `S`, `O`)`.

   Assignment. Return a list of binding conditions, where the first binding condition is the assignment statement itself, and the remaining binding conditions are obtained by back-propagation through the assignment statement, as follows. Let the assignment be
   $Var_n = f(Var_1, Var_2, \ldots, Var_{n-1})$
   Then the variables $Var_1, Var_2, \ldots, Var_{n-1}$ must be bound earlier in the rule, which must therefore contain earlier binding conditions as follows.

If Condition binding Var_1
Condition binding Var_2
...
Condition binding Var_{n-1}
$Var_n = f(Var_1, Var_2, \ldots, Var_{n-1})$

The desired value for Var_n is O, so KRUST back-propagates O through the arithmetic expression f. KRUST solves the equation

$$O = f(Var_1, Var_2, \ldots, Var_{n-1})$$

for each in turn of the $Var_i, 1 \leq i \leq n-1$, while holding the other $Var_{i \neq j}$ constant. In other words, it makes the assumption that only one of the Var_i is incorrect, and calculates the value for this Var_i that will yield the desired value for Var_n. Let the original value held by Var_i be S_i, and the solution to the equation for Var_i be O_i. For each of these Var_i, KRUST then applies `find-binding-condition`(R, Var_i, S_i, O_i) recursively.

At the moment, the equation-solver is a simple one, and will only handle expressions including the operators $+, -, \times$ and $/$, and where each variable occurs only once in the expression.

The procedure `find-binding-condition` just defined returns a list of pairs: binding rule-element and corrected value. Each rule-element is either an OAV where the value is instantiated, or an assignment. Where the condition is an OAV, the corrected value is a replacement for the original value. Where the condition is an assignment, the corrected value is the value that should be returned by the expression on the right-hand side of the assignment. For example, bindings returned by `find-binding-condition` might be.

Rule: DEFAULT-BINDER-LEVEL
Conclusion: set the value of BINDER in FORMULATION to be 0.041
Corrected value: 0.021

Rule: SET-BINDER-CONC

Condition: <ADJUSTED-CONC> = <CONC> / <TOTAL-CONC>

Corrected value: 0.02

KRUST then uses the output of `find-binding-condition` to generate refinements, one for each rule-element. For each element that is a goal, KRUST rewrites it, substituting the corrected value for the original one. For each condition that is an assignment, KRUST calculates a new arithmetic expression for the right-hand side of the assignment. However, there is not enough information in a single refinement example to derive a new arithmetic expression, so KRUST uses values from other examples suffering from similar faults. This is an example of the use of induction, and is described in section 6.1.

The remainder of this section is devoted to an example of how KRUST corrects a wrong-fire rule by correcting the conclusion of one of its antecedent rules; that is, a rule occurring earlier in a rule chain leading to the wrong-fire rule.

5.5.4 An example of the refinement of a wrong-fire rule

This section illustrates how KRUST fixes a particular fault involving a wrong-fire rule. The system and oracle both choose gelatin as the binder, but they differ with respect to the concentration of gelatin required; the system value is 0.041, and the oracle's value is 0.021. The fault submitted to KRUST is therefore

System: GELATIN has value 0.041 in FORMULATION

Oracle: GELATIN has value 0.021 in FORMULATION

The chain of rules leading to this conclusion shown in figure 5.11. These rules interact as follows. Once gelatin has been chosen as binder (by other rules), rule `DEFAULT-BINDER-LEVEL` sets the concentration of gelatin to be 0.04. Rule `SET-BINDER-CONC` then adjusts this value slightly, dividing the concentration by the `TOTAL-CONCENTRATION`, and obtaining a value of 0.041. The rule then writes gelatin and its adjusted concentration onto the `TABLET-REPORT-AGENDA`. Finally, rule `UPDATE-FORMULATION` reads the binder concentration from the agenda and sets the concentration of gelatin in the formulation to be 0.41.

KRUST identifies the rule `UPDATE-FORMULATION` as a wrong-fire rule, so tries either to disable or correct it. In order to correct it, KRUST invokes `find-binding-condition` with arguments

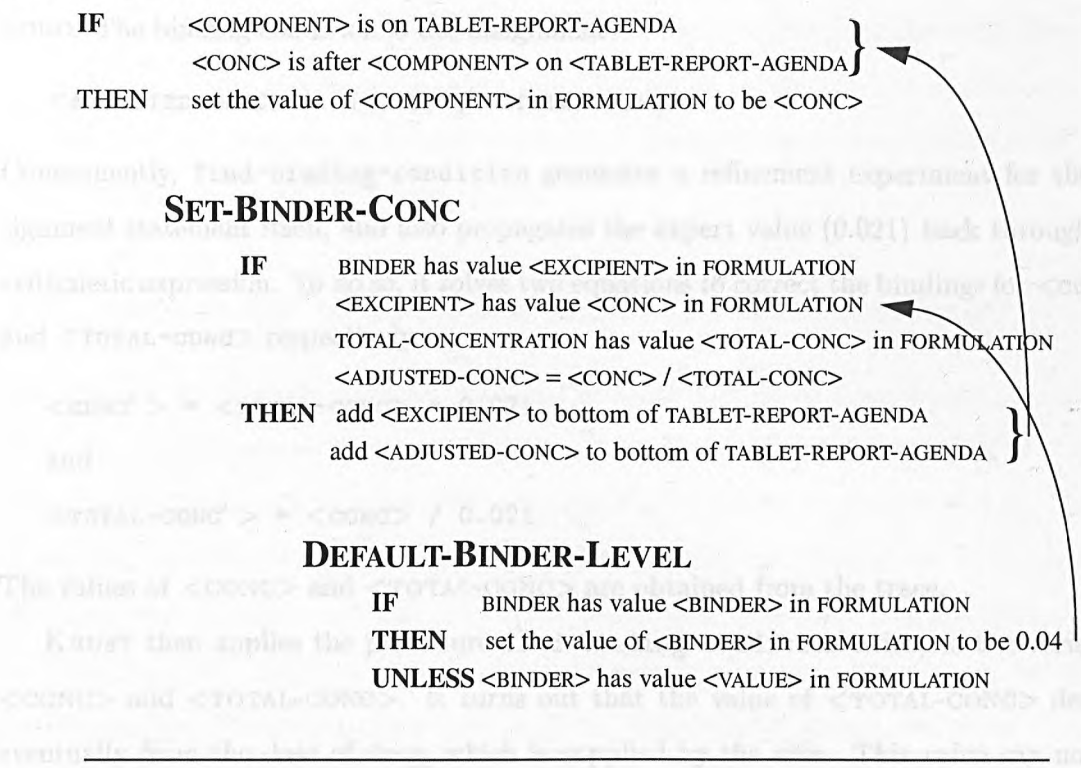

Figure 5.11: Rule chain for wrong binder weight example

Rulename: UPDATE-FORMULATION

Variable: <CONC>

System value: 0.041

Expert value: 0.021

The procedure finds that the binding condition for <CONC> in rule UPDATE-FORMULATION is

```
<CONC> is after <COMPONENT> on TABLET-REPORT-AGENDA
```

This condition is a goal, which chains with the conclusion of rule SET-BINDER-CONC

```
add <ADJUSTED-CONC> to bottom of TABLET-REPORT-AGENDA
```

so `find-binding-condition` is reinvoked recursively with new arguments

Rulename: SET-BINDER-CONC

Variable: <ADJUSTED-CONC>

System value: 0.041

Expert value: 0.021

The procedure next looks for the binding condition for <ADJUSTED-CONC> in SET-BINDER-CONC. The binding condition is the assignment

<ADJUSTED-CONC> = <CONC> / <TOTAL-CONC>

Consequently, `find-binding-condition` generates a refinement experiment for the assignment statement itself, and also propagates the expert value (0.021) back through the arithmetic expression. To do so, it solves two equations to correct the bindings for <CONC> and <TOTAL-CONC> respectively.

<CONC′> = <TOTAL-CONC> * 0.021

and

<TOTAL-CONC′> = <CONC> / 0.021

The values of <CONC> and <TOTAL-CONC> are obtained from the trace.

KRUST then applies the procedure `find-binding-condition` twice, to the variables <CONC> and <TOTAL-CONC>. It turns out that the value of <TOTAL-CONC> derives eventually from the dose of drug, which is supplied by the user. This value can not be changed, so the application of `find-binding-condition` to <TOTAL-CONC> does not lead to a refinement. However, the application of `find-binding-condition` to <CONC> is more successful. KRUST obtains the corrected value for <CONC> by solving the first of the two equations above, having determined from the trace that the actual value of <TOTAL-CONC> is 0.97.

<CONC′> = 0.97 * 0.021

⇒ <CONC′> = 0.02

The binding condition for <CONC> in rule `SET-BINDER-CONC` chains with the conclusion

`set the value of <BINDER> in FORMULATION to be 0.04`

in rule `SET-BINDER-LEVEL`. This conclusion is an OAV with its value bound, so procedure `find-binding-condition` terminates and returns the condition

`set the value of <BINDER> in FORMULATION to be 0.04`

with a corrected value, derived above, of 0.02. The corresponding refinement generated by KRUST is to change the value 0.04 in the rule conclusion to 0.02.

This concludes the discussion of changes required by KRUST in order to apply it to the TFS application. The chapter concludes by considering how general is the work described, and how easy it would now be to apply apply KRUST to a different TFS application. The next chapter will describe additional inductive techniques which were implemented with a view to improving KRUST's performance.

5.6 Generality of the work described in this chapter

This chapter has described the work needed to apply KRUST to a particular application, TFS. However, because of the generic approach taken during the development of KRUST, in particular, the common knowledge representation used for representing rules, KRUST is now to a large extent applicable to *any* TFS application. There will be no need to adapt the core routines of blame allocation, refinement generation and implementation, and filtering when applying KRUST to a new such application. However, a number of application-specific modifications *were* made to KRUST during its adaptation to TFS, and attention was drawn to these earlier in this chapter. These are the aspects of KRUST which would have to be re-implemented.

The rule translator. This is in all respects but one a PFES translator, rather than a TFS translator. The only TFS-specific elements are the rules which translate agenda operations. These work by recognising certain particular sequences of operations within the PFES rules. To do so, they require domain-specific knowledge specifying which attributes are numeric, and which are symbolic. To adapt the translator to a new application, it would be necessary to provide it with knowledge about the attribute types in that application. It might also be necessary to add new rules to recognise different sequences of agenda operations.

The translator is written "defensively", so that if it encounters a sequence of agenda operations which it does not recognise, it will signal an error and terminate. This indicates to the knowledge engineer that new rules must be added to the translator to handle the new situation. Similarly, if knowledge about attribute types is absent, then the translator may not be able to translate certain agenda operations, and will again signal an error and terminate. The defensive approach was adopted on the

grounds that is preferable for the translator to signal an error when encountering a novel situation, rather than to proceed and risk generating incorrect output.

Input and output. The TFS application has three inputs: drug, dosage, and number of fillers. When KRUST was adapted for use with PFES, two modifications were made.

- The KBS testing module of KRUST was modified so that, when it ran a refined KBS on a given formulation problem, it wrote the problem inputs to a file.
- The PFES shell was modified so that it read its input from the above file, rather than from the user via a GUI.

This interface would not work unchanged for a different PFES application, with different input variables. In order to run KRUST on a new PFES application, it would be necessary to make minor adjustments to the above features of the KBS testing module and the PFES shell, so that the problem inputs were written to file, and then read and assigned to the appropriate PFES variables.

Chapter 6

Induction

This chapter presents two applications of induction within the KRUST system. The more significant of these, which constitutes the bulk of the chapter, is the use of inductive operators for refinement implementation. The second application uses induction for refinement filtering. The applications share a common approach: that of learning from the traces of multiple related examples.

The new inductive operators learn from multiple training examples, permitting KRUST to generate more accurate refinements, and to fix faults which it previously could not have fixed. Chapter 4 identified the absence of inductive refinement operators as a significant weakness in KRUST, and sections 5.5.1 and 5.5.3 drew attention to situations where refinement could be facilitated by the use of information derived from multiple training examples. These observations provide motivation for the work described in this chapter.

In the absence of inductive operators, KRUST generates refinements for a single refinement example at a time, subsequently using other examples for filtering and judging. This chapter introduces a new approach whereby KRUST constructs *sets* of *related* examples, together with their traces, and then uses these examples and traces to constrain the refinement implementation process. In this way, KRUST is able to make use of training examples other than the refinement example at an earlier stage than before. The approach may be summarised as follows.

- Select as positive examples those examples and their traces that exhibit a particular fault, and as negative examples those that do not.
- Identify features distinguishing these two groups.

- Use these features to guide refinement implementation.

The following inductive operators are described.

inductive_change_formula uses curve-fitting to correct an arithmetic expression in a rule condition (section 6.1).

inductive_add_fact adds a new fact to the database (section 6.2). It is an improved form of the **add_fact** operator described in section 5.5.1.

inductive_adjust_value adjusts the threshold value in a comparison condition. It is an improved form of the **adjust_value** operator used by the original KRUST.

inductive_split_rule makes two or more copies of a rule, and adds extra conditions to some of the copies, thus changing the order in which the rules select excipients (section 6.4).

The technique of refinement filtering uses a similar approach, but for a different purpose (section 6.5). As with the inductive operators, features are identified which distinguish faulty from non-faulty traces. However, the task for which these features are used is to identify rules which could not be responsible for the fault, and so filter out refinements to these rules.

A summary of the work described in this and the previous chapter has been published as (Boswell, Craw & Rowe 1997). A shorter summary, giving greater emphasis to KRUST's representation of knowledge and proofs, and to the role of refinement in software development, has been published as (Craw, Boswell & Rowe 1997).

6.1 Learning formulae from multiple examples

This section describes how the operator **inductive_change_formula** can alter an arithmetic expression occurring in a rule condition. This operator complements work described in the previous chapter. Section 5.5.3 stated that a wrong-fire rule might be corrected by changing either a rule conclusion or an arithmetic expression, and went on to explain how a rule conclusion can be corrected, but postponed discussion of arithmetic expressions,

since their correction requires induction. The current section describes the use of induction in fixing arithmetic expressions, and thus completes the account of how KRUST fixes wrong-fire rules.

Recall that the function **find-binding-condition** works backwards from an incorrect result and returns a list of rule elements at which the error may have originated (section 5.5.3). As just stated, these can be either rule conclusions or assignments. The operator **inductive_change_formula** is invoked whenever **find-binding-condition** returns an assignment to an arithmetic condition, for example

$$y = x \times x + 1$$

Currently, the operator is applicable only to expressions including just one variable on the right-hand side of the assignment (x in this example). The reason for this is that the operator uses curve-fitting to learn a new function, and it is not thought feasible to use this technique to learn a function of more than one variable, because of the size of the search space.

When **inductive_change_formula** is applied to a condition $y = f(x)$, it performs the following steps, each of which is described in greater detail later.

1. Select sets of positive examples, which are defined to be those training examples which exhibit the same fault as the refinement example, and negative examples, which do not exhibit the fault.

2. For each positive example e, determine the value y which will lead to the expert solution, and the value to which x was bound. Let these values be y_e and x_e respectively.

3. Determine a threshold value x_0 for x which separates positive from negative examples. (Situations where no such threshold exists are discussed in the more detailed account).

4. Apply a curve fitting technique to the pairs (x_e, y_e) to derive a function g such that $y_e \approx g(x_e)$ for all e.

5. Replace the faulty rule with a pair of new rules. Each new rule includes an added

condition comparing x with the threshold calculated in step 3., so that one will fire for positive examples and one for negative examples. The rule for negative examples is otherwise unchanged. The rule for positive examples has the condition $y = f(x)$ replaced with a new condition $y = g(x)$. The effect of this change is that the refined KB will treat negative examples of the fault as before, but will apply the new equation to positive examples of the fault. The following rule template illustrates the process, for the case where the positive examples have $x \geq x_0$.

Old rule

If

...

$y = f(x)$

...

Then

...

New rules

If	If
...	...
$x < x_0$	$x \geq x_0$
$y = f(x)$	$y = g(x)$
...	...
Then	Then
...	...

Currently the application of the **inductive_change_formula** is restricted in the following way. At step 2, KRUST will *not* back-propagate values through one arithmetic expression in order to correct an arithmetic expression occurring earlier in the chain. For example, suppose KRUST seeks to correct the following sequence of rule conditions, each consisting of an assignment.

$$\begin{aligned} x &= 2 \\ y &= x^2 + 1 \\ z &= 2 \times y \end{aligned}$$

These lead to the result $z = 10$. Suppose the desired result is that $z = 20$. Then to correct this result, KRUST will generate refinements in the following ways (figure 6.1)

- Apply **inductive_change_formula** to the final condition $z = 2 \times y$. (Refinement 1 in figure 6.1).
- Back-propagate the value $z = 20$ through the two equations to the setting condition $x = 2$ and replace this condition with a new, corrected, condition $x = 3$. (Refinement 2 in figure 6.1).

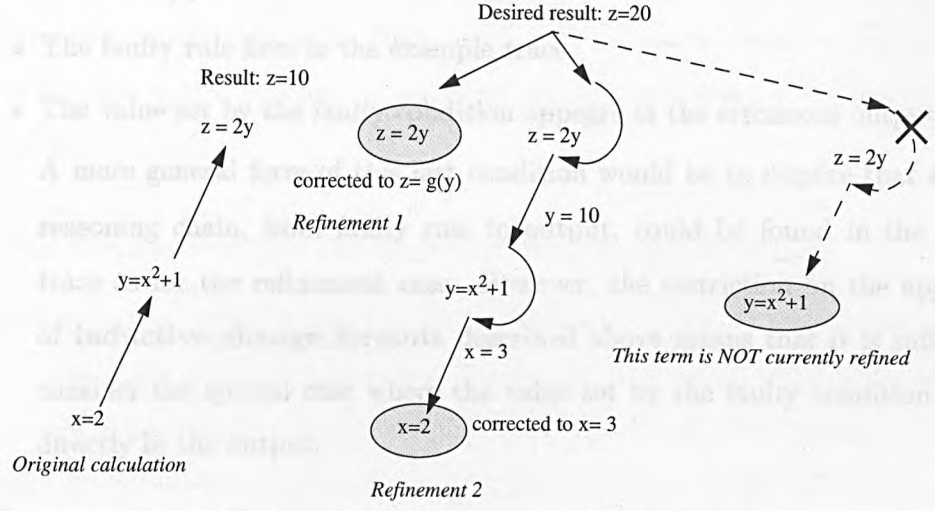

Figure 6.1: Refinements made to chained calculations

However, KRUST will *not* back-propagate the value through the last equation $z = 2 \times y$ so that **inductive_change_formula** can be applied to the condition $y = x^2 + 1$. (dotted lines in figure 6.1). The reason for this lies in the separation of the refinement generation process from the refinement implementation process. Back-propagation is carried out at the refinement generation stage. Therefore, if **inductive_change_formula** were to be applied to an intermediate condition such as $y = x^2 + 1$, it would need to repeat the back-propagation process (the dotted path in the figure) for all the other positive examples. This would require that the refinement generation procedure pass details of all intermediate calculations to the implementation operator, to enable it to perform the back-propagation process itself. There is currently no mechanism in KRUST for this information to be passed

to an implementation operator, through it would be feasible to expand the current data structures to permit it.

The four steps taken by **inductive_change_formula** are now described in more detail. In this description, the 'faulty condition' is the assignment statement to which **inductive_change_formula** is being applied, and the 'faulty rule' is the rule containing this condition. Field F is the field of the system output which is incorrect, and which KRUST is currently trying to fix.

1. **Select example sets.** The positive examples of the fault are characterised by the following properties.

   - The example shows an error in the field F.
   - The faulty rule fires in the example trace.
   - The value set by the faulty condition appears in the erroneous output.

   A more general form of this last condition would be to require that the same reasoning chain, from faulty rule to output, could be found in the example trace as for the refinement case. However, the restriction on the application of **inductive_change_formula** described above means that it is sufficient to consider the special case where the value set by the faulty condition appears directly in the output.

   Negative examples of the fault are characterised by the following properties.

   - The example shows a correct value in field F.
   - The faulty rule fires in the example trace.
   - The value set by the faulty condition appears in field F.

   The negative examples are needed if the faulty rule is firing correctly in some circumstances and incorrectly in others. If the faulty rule always fires incorrectly, then no negative examples will be found.

2. **Determine x_e and y_e for each example.** The value of x_e can be determined from the trace. Since there is a direct sequence of variable unifications between y_e and the output (i.e., as explained above, KRUST will not back-propagate through arithmetic expressions), the value of y_e is currently obtained directly from the oracle's output.

3. **Determine threshold.** If there are no negative examples, then this step is omitted. If negative examples *do* exist, the operator currently requires that a threshold value must separate positive from negative examples. If positive and negative examples are mingled in some more complex way, so that no such threshold exists, then the operator will not generate a refined KB.

4. **Curve fitting.** The technique used is that of least squares polynomial approximation. This requires the solution of $n+1$ simultaneous equations in $n+1$ unknowns, which is carried out by Gaussian elimination (Atkinson & Hartley 1983). Currently, the polynomials used are cubics.

5. **Create new rules.** This process has already been explained in the earlier outline. One special case needs to be dealt with: if there are no negative examples, then just one new rule is created, by replacing the old expression $y = f(x)$ with a new expression $y = g(x)$.

6.1.1 An example of the use of inductive_change_formula

This operator proves useful in fixing a particular fault in TFS. The fault occurs in examples for which the value of dose is high, and its symptoms are that many attributes in both specification and formulation are incorrect (figure 5.4). The method of selecting independent attributes described in section 5.2 indicates that all the incorrect attributes are dependent on the attribute TARGET-TABLET-WEIGHT, so a typical input to KRUST for this fault is

System: TARGET-TABLET-WEIGHT OF SPECIFICATION == 400

Oracle: TARGET-TABLET-WEIGHT OF SPECIFICATION == 450

KRUST determines that the rule 1ST-GUESS-WEIGHT is wrong-fire.

```
RULE 1ST-GUESS-WEIGHT
IF    DRUG has value <DRUG> in the FORMULATION
 AND  <DRUG> has value <DOSE> in the FORMULATION
 AND  <WEIGHT> = ROUND-TO-NEAREST-5(100 * <DOSE>
                 /(0.221 * <DOSE> + 10))
THEN
```

```
set the value of TARGET-TABLET-WEIGHT in
                SPECIFICATION to be <WEIGHT>
```

Within this rule, the value of TARGET-TABLET-WEIGHT is set by the condition

```
<WEIGHT> = ROUND-TO-NEAREST-5(100 * <DOSE>
          /(0.221 * <DOSE> + 10))
```

KRUST therefore applies the operator **inductive_change_formula** to this condition. Its aim is trying to find the correct equation which relates <WEIGHT> to <DOSE>; these correspond to y and x respectively in the algorithm outline. Its first step is to select positive and negative examples. Positive examples are those for which the rule 1ST-GUESS-WEIGHT fires and calculates an incorrect value for TARGET-TABLET-WEIGHT. It turns out that the value of <DOSE> (x_e) is 360 for all positive examples, and the value of <TARGET-TABLET-WEIGHT> (y_e) is 450. Negative examples are those for which 1ST-GUESS-WEIGHT fires and calculates a correct value for TARGET-TABLET-WEIGHT. The values taken by dose in the negative examples cover all the values in the example set less than 360, viz., 310, 260, 210 ... 10. (This "example set" referred to here is one that was generated in order to evaluate the effectiveness of KRUST. Inputs for the examples were selected in such a way as to cover evenly the space of possible inputs. This process is described in detail in section 8.1.1).

It appears that a step of 50 between values of <DOSE> was too coarse for refining this particular fault, so further examples were generated with <DOSE> varying from 310 to 360 in a step size of 1. (TFS will not create formulations when the dose is greater than 360). This experiment had two conclusions.

- Positive examples of the fault occur when the dose is greater than 350.
- The correct target-tablet-weight when the dose lies in the range 351 to 360 is a constant 450.

Consequently, the use of extra examples allowed KRUST to determine the threshold between correct and incorrect rule behaviour more precisely, but not to induce an interesting" polynomial. The curve-fitting routine naturally calculated the new function g to be

```
RULE RULE1
IF    DRUG has value <DRUG> in FORMULATION
 AND  <DRUG> has value <DOSE> in FORMULATION
 AND  <DOSE> is greater-equal 351
 AND  <WEIGHT> = 450 + 0 * <DOSE> + 0 * <DOSE>^2
              + 0 * <DOSE>^3
THEN
 set the value of TARGET-TABLET-WEIGHT in
              SPECIFICATION to be <WEIGHT>

RULE RULE2
IF    DRUG has value <DRUG> in FORMULATION
 AND  <DRUG> has value <DOSE> in FORMULATION
 AND  <DOSE> is less-than 351
 AND  <WEIGHT> = (ROUND-TO-NEAREST-5(100 * <DOSE>
              /(0.221 * <DOSE> + 10))
THEN
 set the value of TARGET-TABLET-WEIGHT in
              SPECIFICATION to be <WEIGHT>
```

Figure 6.2: Two rules learned by **inductive_change_formula**

$$450 + 0 * \langle\text{DOSE}\rangle + 0 * \langle\text{DOSE}\rangle^2 + 0 * \langle\text{DOSE}\rangle^3$$

The values of dose available in the original positive and negative examples implied a threshold value of dose somewhere between 310 and 360. The threshold was chosen within that range so as to minimize the change to the KB; in other words, to minimize the number of occasions on which the new function will be used. The value chosen was therefore 360. For the extended example set, a more accurate threshold of 351 was obtained. Thus the effect of the refinement, when using the extended example set, was to replace the rule 1ST-GUESS-WEIGHT by the two rules shown in figure 6.2

The domain expert subsequently informed me that the correct formula for target-tablet-weight, for high values of dose, included a cubic polynomial, but that the value calculated by the polynomial was then rounded to the nearest 5mg. This explains why, for a narrow range of doses, only a single value of target-tablet-weight was obtained.

6.2 Using traces to learn facts

The example in section 5.4.7 showed a situation in which a new fact was to be learned, but where the refinement example on its own provided few constraints on the new value. This section describes the operator **inductive_add_fact**, which uses information from other training examples to provide further constraints on the new fact. As in the previous section, the manner in which the operator works is first described briefly, then each of the steps is discussed in more detail. Finally, an example is given where the operator corrects a fault in TFS.

inductive_add_fact is similar in application to the **add_fact** operator described in section 5.5.1. It implements the refinement instruction to generalise a condition. It is applied only to conditions which are OAV Triples, where the triple refers to an attribute of a type that is stored in the fact database. Examples of such conditions are

```
SOLUBILITY of <DRUG> is <DRUG-SOLUBILITY>
YP of <FILLER> is <YP-SLOW>
YP-FAST of <DISINTEGRANT> is <DISINTEGRANT-YP-FAST>
```

Let the unsatisfied condition be C, and the rule containing this condition be R. C is necessarily an OAV triple, and its object element will usually be a variable. Let the value taken by that variable when the condition fails for the refinement case be O. (If the object is a constant, then let O be that constant). O is thus the object for which a fact is missing from the database. In the case of TFS, O is always either an excipient or a drug. For example, if the unsatisfied condition is

```
YP of <FILLER> is <YP-SLOW>
```

which fails when <FILLER> is calcium phosphate, then the object in the OAV Triple is <FILLER>, and O is calcium phosphate. **inductive_add_fact** performs the following steps.

1. Select sets of positive examples, which exhibit the same fault as the training example, and negative examples, which do not exhibit the fault.

2. Derive bounds on the value to be learned from the two sets of examples. The bounds derived from the positive examples are selected so that the new fact should enable

the previously unsatisfied rule R for each positive example. The bounds derived from the negative examples are selected so that the new fact should *not* enable R for any of the negative examples.

3. Choose a value V that lies within the bounds. If the bounds do not determine a unique value, choose a value such that adding the new fact will constitute the most conservative change to the KB.
4. Add the new fact to the database.

The most complex of these steps is the first: the selection of positive and negative examples. The assumption behind the addition of the new fact about object O is that, for the refinement example, rule R failed to fire for O, but should have fired. The evidence that R should have fired is that object O appears in the oracle's formulation but not in the system's. Hence, positive examples of the fault must satisfy the following conditions.

- Condition C in rule R failed for object O, and prevented rule R firing for object O.
- Object O appears in field F of the oracle's formulation.
- Object O does not appear in field F of the system's formulation.

It is of course possible that, for some example, rule R will fail incorrectly for some object O, but that if it had succeeded, object O would have been correctly filtered out at some later stage. In that case, the behaviour of the system as a whole will be correct, since the faulty behaviour of R is masked by the later rule, so the example can not be identified as a positive example of the fault.

Negative examples on the other hand, are those for which rule R correctly fails to fire for object O. The correctness of R's failure to fire for O may be determined from the observation that object O does *not* appear in the oracle's formulation. Thus negative examples must satisfy the following conditions.

- Condition C in rule R failed for object O, and prevented rule R firing for object O.
- Object O does not appear in field F of the oracle's formulation.

There is the possibility of one further complication which requires an alteration to these conditions. Until now, only monotonic rules have been considered, where the firing

of one rule can enable a subsequent rule, but can never disable it. However, there are conclusions in PFES which are non-monotonic. For example, if the conclusion

`remove-from-agenda <AGENDA> <ITEM>`

is executed with <ITEM> bound to, say TALC, it will *prevent* the condition `on-agenda <AGENDA> <ITEM>` from firing for TALC.

If the rule-chain from the rule R includes a non-monotonic rule, then the positive and negative examples must be redefined. The assumption that a missing fact prevents rule R from firing is unchanged, but the effect on the KB's output, and hence the definition of positive and negative examples, is altered. As before, positive examples of the fault are those for which the failure of condition C prevented rule R from firing for object O. But the effect is reversed; it causes O to be recommended in the formulation when it should not have been. Hence a positive example must now satisfy these conditions.

- Condition C in rule R failed for object O, and prevented rule R firing for object O.
- Object O does not appear in field F of the oracle's formulation.
- Object O appears in field F of the system's formulation

The conditions for negative examples are now.

- Condition C in rule R failed for object O, and prevented rule R firing for object O.
- Object O appears in field F of the oracle's formulation.
- Object O appears in field F of the system's formulation.

This concludes the description of how **inductive_add_fact** selects positive and negative examples. The remaining steps are more straightforward, but some details remain to be explained.

2. **Deriving bounds on the new fact.** Since the positive examples are examples of the same fault as the refinement example, they constrain the new fact in the same way. Suppose the rule condition matching the missing fact is

   attribute of *object* is A

   and a subsequent rule condition including A is $A < t$. Then *attribute* of *object* is chosen so that the rule will fire, so it has to satisfy the condition

attribute of *object* $< t$

Hence a condition which is an upper bound leads to an upper bound on the new fact, and similarly for lower bounds. On the other hand, negative examples are those for which the rule correctly failed to fire, so the new fact must be chosen in such a way that the subsequent condition is *not* satisfied. Hence the direction of the bounds are reversed: given the same condition $A < t$, the negative examples impose a lower bound on A.

3. **Choosing the most conservative value.** The most conservative value is the value whose addition constitutes the minimal change to the KBS. The way this is interpreted in the case of a new fact is to choose a value which causes a minimal change to the rule currently being refined; that is, the rule which failed to fire for a particular object because of a missing fact. The most conservative value is thus the one which minimises the occasions on which the rule will fire, while still causing it to fire for all the positive examples. If the subsequent condition on A is an upper bound, then the most conservative value for A is the highest value lying within the bounds derived from the examples; if the condition is a lower bound, then the most conservative value is the lowest value lying within the bounds.

6.2.1 An example of the use of inductive_add_fact

The use of **inductive_add_fact** is now illustrated with reference to the fault described earlier in section 5.4.7, in which a missing fact causes TFS to recommend the filler Calcium Phosphate incorrectly. For convenience, the figure illustrating the rule chain is reproduced again here (figure 6.3).

As before, KRUST determines that the condition

```
MAX-LEVEL has value <LEVEL> in <FILLER>
```

fails to fire for calcium phosphate, because the max-level of calcium phosphate is missing from the database. In the terms in which the algorithm was described above, the object O is calcium phosphate, and the attribute A for which a value is missing from the database is MAX-LEVEL.

The **inductive_add_fact** operator, which replaces **add_fact**, proceeds as follows. It first selects positive and negative examples. Since the rule-chain contains the non-

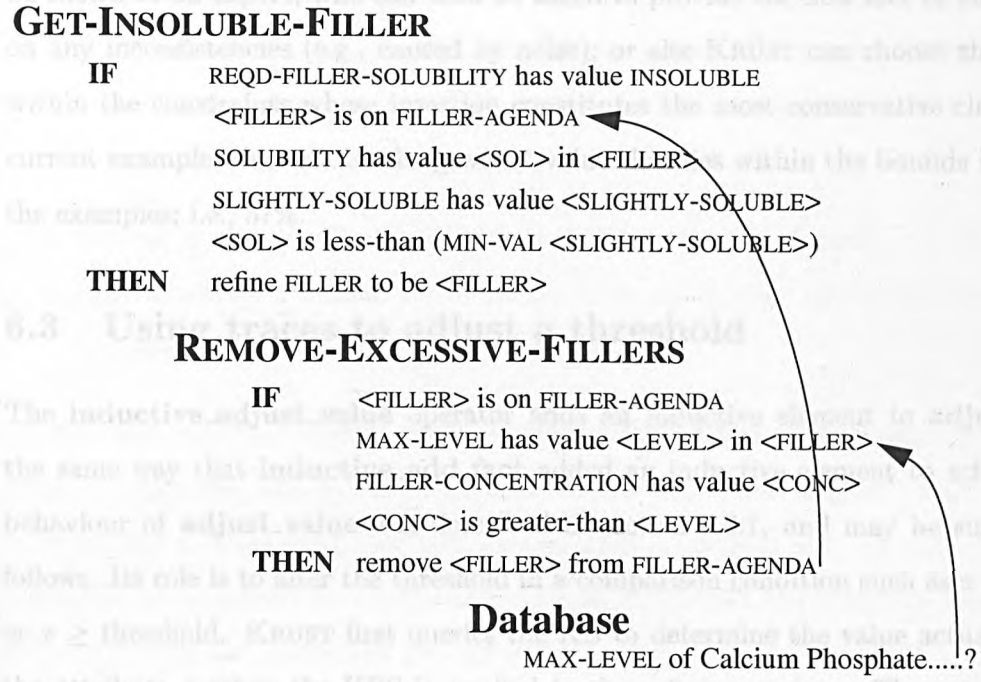

Figure 6.3: Rule chain for wrong filler example

monotonic conclusion

```
remove <FILLER> from FILLER-AGENDA
```

the second pair of definitions are used. Positive examples are those for which REMOVE-EXCESSIVE-FILLERS failed to fire for calcium phosphate, and where calcium phosphate appears as filler in the system formulation but not the oracle formulation. Negative examples are those where again REMOVE-EXCESSIVE-FILLERS failed to fire for calcium phosphate, but where calcium phosphate appears in both the system and oracle formulations.

The positive examples then impose an upper bound on the max-level of calcium phosphate, in just the same way as the refinement case does. (section 5.5.1). Conversely, the negative examples impose a lower bound on the max-level. This is because REMOVE-EXCESSIVE-FILLERS correctly failed to fire for the negative examples; the value of max-level must be chosen to be sufficiently high that it does not cause the rule to start firing for these examples.

In practice, the available examples constrain the max-level of calcium phosphate to lie between 11.5% and 58%. The constraints can be used in one of two ways: they can either

be shown to an expert, who can then be asked to provide the new fact or else adjudicate on any inconsistencies (e.g., caused by noise); or else KRUST can choose the value from within the constraints whose insertion constitutes the most conservative change. In the current example, this will be the greatest value that lies within the bounds induced from the examples; i.e., 57%.

6.3 Using traces to adjust a threshold

The **inductive_adjust_value** operator adds an inductive element to **adjust_value** in the same way that **inductive_add_fact** added an inductive element to **add_fact**. The behaviour of **adjust_value** was described in section 4.6.1, and may be summarised as follows. Its role is to alter the threshold in a comparison condition such as $x <$ threshold, or $x \geq$ threshold. KRUST first queries the KB to determine the value actually taken by the attribute x when the KBS is applied to the refinement case. Then, to *specialise* a condition, it adjusts the threshold in the rule condition just far enough to *prevent* the condition from being satisfied; alternatively, to *generalise* the condition, it adjusts the threshold just far enough to *allow* the condition to be satisfied.

The **inductive_adjust_value** operator uses information derived from additional examples to determine the best adjustment to the threshold value in a condition. It works in a similar way to **inductive_add_fact**, though one difference is that **inductive_adjust_value** can be used either to generalise or to specialise a condition in some rule R. It performs the following steps.

1. Select sets of positive examples, which exhibit the same fault as the training example, and negative examples, which do not exhibit the fault.
2. Derive bounds on the new value of the threshold from the two sets of examples. The precise role of positive and negative examples depends on whether the operator is generalising or specialising a condition. If generalising, then the bounds derived from the positive examples are selected so that the new fact should enable the previously unsatisfied rule R for each positive example. The bounds derived from the negative examples are selected so that the new fact should *not* enable R for any of the negative examples. On the other hand, if the operator is specialising a condition, then the

bounds derived from the positive examples are selected so that the new fact should disable the previously satisfied rule R for each positive example. The bounds derived from the negative examples are selected so that the new fact should *not* disable R for any of the negative examples.

3. Choose a new threshold value T that lies within the bounds. If the bounds do not determine a unique value, choose T to be as close as possible to the original threshold.

The only step that needs further explanation is the first: the selection of positive and negative examples. The details depend on whether **inductive_adjust_value** is being used to generalise a unsatisfied condition or specialise a satisfied one. In either case, let the condition being refined be condition C in rule R, and let the attribute whose incorrect value is driving the refinement process be A.

Generalisation. Here positive examples of the fault are those for which

- the system output for attribute A is incorrect, and
- rule R failed because condition C was unsatisfied.

Negative examples of the fault are those for which

- the system output for attribute A is correct
- rule R failed because condition C was unsatisfied.

When generalising, the operator attempts to adjust the threshold in condition C so that it is satisfied for the positive examples but not for the negative examples.

Specialisation. Here positive examples of the fault are those for which:

- the system output for attribute A is incorrect, and
- rule R succeeded.

Negative examples of the fault are those for which

- the system output for attribute A is correct
- rule R succeeded.

When specialising, the operator attempts to adjust the threshold in condition C so that it is no longer satisfied for the positive examples, but remains satisfied for the negative examples.

6.3.1 An example of the use of inductive_adjust_value

This operator proves useful in fixing a particular fault in TFS-1B. This fault is similar to the fault in TFS-1A where, for certain values of dose, an incorrect value of target-tablet-weight is calculated. In TFS-1B, the following two rules are used to calculate target-tablet-weight.

```
RULE 1ST-GUESS-WEIGHT-<=350MG
IF    DRUG has value <DRUG> in the FORMULATION
 AND  <DRUG> has value <DOSE> in the FORMULATION
 AND  <DOSE> is less-equal 350
 AND  <WEIGHT> = (ROUND-TO-NEAREST-5(100 * <DOSE>
                 /(0.221 * <DOSE> + 10))
THEN
 set the value of TARGET-TABLET-WEIGHT in
                 SPECIFICATION to be <WEIGHT>

RULE 1ST-GUESS-WEIGHT->350MG
IF    DRUG has value <DRUG> in the FORMULATION
 AND  <DRUG> has value <DOSE> in the FORMULATION
 AND  <DOSE> is greater-than 350
 AND  <WEIGHT> = ROUND-TO-NEAREST-5(3.1317 + 0.375 * <DOSE>
                 - 0.000544 * <DOSE>^2 + 2.53 * 10^-7 * <DOSE>^3)
THEN
 set the value of TARGET-TABLET-WEIGHT in
                 SPECIFICATION to be <WEIGHT>
```

The effect of this pair of rules is to calculate target-tablet-weight using the formula in

the first rule for values of dose less than or equal to 350mg, and to use the second rule for larger values of dose. However, the value of target-tablet-weight calculated by these rules is incorrect for certain values of dose around 310mg. When presented with such an example, KRUST classifies the first rule as wrong-fire and the second rule as no-fire. The refinements generated for the KB include an experiment to specialise the condition <DOSE> $\leq$ 350 in the first rule and to generalise the condition <DOSE> $>$ 350 in the second rule. In both cases, the **inductive_adjust_value** operator is applied. The result of the two applications is to adjust the threshold value from 350 to 310 in both rules. The effect of this change is that the second formula for target-tablet-weight comes into effect at a dose of 311mg, instead of 351mg.

6.4 Using traces to learn new rule conditions

This section first explains the motivation for introducing an operator that learns new conditions, and the nature of the problem which such an operator is intended to solve. It then explains how the operator works, and uses a TFS example to illustrate the kind of refinement produced by the operator.

The motivation for creating an operator to learn new rule conditions was two-fold. First, KRUST's inability to do this has been identified as a weakness in comparison with EITHER. Without such an operator, KRUST can specialise a rule only by specialising a condition within it, or removing the whole rule. Consequently, there are some faults which KRUST can not fix. Secondly, such an operator is required to enable KRUST to carry out automatically the transformation of TFS-1B into TFS-2. The reason is that TFS-2 incorporated a policy change whereby each excipient was assigned a numeric category, and low-category excipients were preferred to high category ones, as described in section 3.3.3. Such a change can most simply be implemented by adding conditions to existing rules: conditions which test an excipient's category, and perhaps other attributes. Therefore a new operator is required which is capable of learning new rule conditions. The operator should be designed so that it is capable of upgrading TFS-1B as described, but not written to solve only that particular problem.

An operator **inductive_split_rule** has been implemented to satisfy these requirements. It uses the inductive approach outlined at the beginning of this section, and taken

by the inductive operators already described. **inductive_split_rule** is applicable when the system formulation contains an incorrect object. In the case of TFS, this object will be an excipient. KRUST's existing refinement procedures will generate specialisation experiments for all rules whose firing contributed to the incorrect conclusion. **inductive_split_rule** can therefore be treated as a specialisation operator. However, it is applicable only to rules which fired for the incorrectly-recommended object. For example, **inductive_split_rule** is applicable to the rule MOST-STABLE-BINDER if and only if the rule fires with variable <BINDER> bound to the value of binder which appears in the system formulation, and that value is incorrect.

```
Rule MOST-STABLE-BINDER
IF    <BINDER> is on STABILITY-AGENDA
 AND  check-is-a <BINDER> BINDER
THEN
 set the value of BINDER in FORMULATION to be <BINDER>
UNLESS
 BINDER has value <VALUE> in FORMULATION
```

Given a faulty rule R, the operator performs the following actions. As before, the procedure is first given in outline, then each of the steps is described in more detail.

1. Select positive and negative examples of the fault.
2. Identify the variable V in the faulty rule which is bound to an object which appears in the system formulation.
3. For each positive example of the fault, let the object to which V is bound be a positive example of a faulty object. For each negative example of the fault let the object to which V is bound be a negative example of a faulty object.
4. For each positive and negative example, create an attribute vector containing all attributes of the formulation problem. In the case of TFS, these will be drug properties, dosage, number of fillers, together with properties of the object (excipient) identified in step 3.

5. Run an attribute-based learning program on the attribute vectors, to determine properties which distinguish positive from negative objects. Currently KRUST uses ID3 (Quinlan 1986).

6. Identify in the output of the learning program those conditions which select negative examples. These are negative examples of the *fault*, so are in fact correctly recommended objects.

7. Make a copy R_1 of rule R, with two modifications: the priority of R_1 is 1 greater than that of R, and R_1 contains the extra conditions calculated in step 6. If the conditions are disjunctive, it will be necessary to make several copies R_i of R, one for each disjunct.

The effect of the additional rule(s) will be that the set of rules $R \cup R_i$ will fire for the same objects as the original rule R, but that the preferred objects (negative examples) will be chosen before the others. The steps of the above procedure are now described in more detail.

1. **Select examples.** Given a refinement case in which rule R fires with V bound to an object O which appears in field F of the formulation, and is incorrect, positive examples of the fault are those which share all these properties of the refinement case. Negative examples of the fault are those in which rule R fires with V bound to an object O which appears in field F of the formulation, and is correct.

2. **Identify variable V.** This was recorded as a separate step in order to clarify the earlier description of the algorithm, but in practice the variable V is identified in the course of step 1.

3. **Identify positive and negative objects.** This requires no further comment.

4. **Create attribute-vectors.** The attributes used to construct attribute vectors are formed from the problem input, the properties of all objects in the problem input, the object chosen by rule R, and the properties of that object. These constitute the 'observables' in the formulation problem. In the case of a TFS problem, the following attributes are included.

The problem input. This consists of the drug, dose, and number of fillers. However, the drug itself is *not* included in the attribute vector, though its properties are. The reason is that TFS is intended to design tablets for any drug by taking into account the chemical properties of that drug; it does not make sense to include rules which fire only for some particular named drug. Therefore, only the dosage and number of fillers are included in the attribute vector.

Properties of objects in the problem input. The drug *properties* are included in the attribute vector.

The object chosen by rule R. This will be an excipient.

Properties of the object. Properties of the excipient.

5. **Induction.** The next step consists of running ID3 on the attribute vectors. The resulting decision tree is expressed in Lisp as a nested list, which may be conveniently processed by KRUST.

6. **Condition selection.** The operator identifies in the output of the learning algorithm those conditions which select negative examples. These will be formed from the various paths from the root of the decision tree to the negative leaf nodes, and will thus be in disjunctive normal form, $C_1 \vee C_2 \ldots C_n$, where $C_i = c_1 \wedge c_2 \ldots c_{m_i}$.

   For example, given the decision tree shown in figure 6.4, the following two conjuncts will be selected.

```
STABILITY of DRUG is greater-than 10
SRS of BINDER is less-than-equal 50

STABILITY of DRUG is less-than-equal 10
YPS of BINDER is greater-than 21
```

7. **Creating new PFES rules.** The conditions derived from the decision tree can not in general be inserted directly into PFES rules, since many of the branching conditions have to be represented by multiple PFES conditions. The difficulty lies in the fact that PFES requires two conditions to test the value of an attribute: one to store the value in a variable, and one to test the value of the variable. For example, testing that the srs of talc is higher than some threshold requires the two conditions

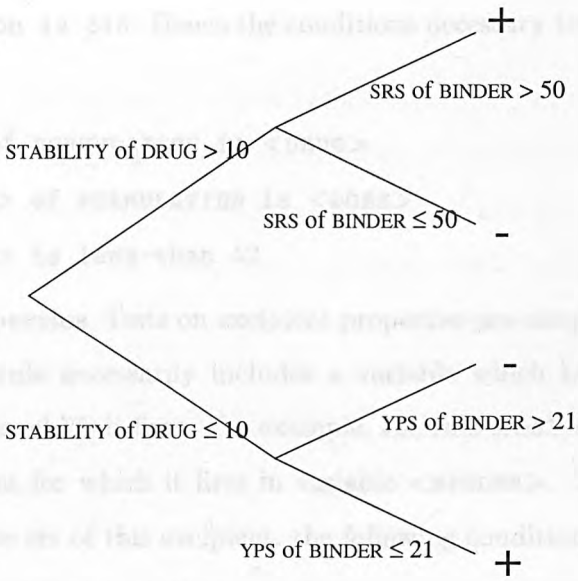

Figure 6.4: Decision tree induced by ID3

SRS of TALC is <SRS>

<SRS> is greater-than *threshold*

The details of how conditions are represented depends on the attributes concerned. For each type of attribute, the list below illustrates how a test of the value of the attribute can be as a sequence of TFS conditions. In each case, the comparison chosen is "value is less-than 42", which should be understood as representing a typical comparison.

No-of-fillers. This condition can be represented as follows

NO-OF-FILLERS of FORMULATION is <NUMBER>

<NUMBER> is less-than 42

Drug properties. A drug property, such as solubility, can be tested as follows.

DRUG of FORMULATION is <DRUG>

SOLUBILITY of <DRUG> is <SOLUBILITY>

<SOLUBILITY> is less-than 42

Dosage is a special case of a drug property, since it is represented in TFS as the value of the drug attribute. In other words, if the drug is DRUG-A, and the dose is 310, then this will be represented by the OAV Triple DRUG-A of

formulation is 310. Hence the conditions necessary to test the value of dose are

```
DRUG of FORMULATION is <DRUG>
<DRUG> of FORMULATION is <DOSE>
<DOSE> is less-than 42
```

Excipient properties Tests on excipient properties are simplified by the fact that the faulty rule necessarily includes a variable which holds the name of the excipient for which it fires. For example, the rule MOST-STABLE-BINDER stores the excipient for which it fires in variable <BINDER>. Therefore, to test, for example, the srs of this excipient, the following conditions are used.

```
SRS of <BINDER> is <SRS>
<SRS> is less-than 42
```

Excipient stabilities. These are represented in TFS as an agenda, rather than as attributes, so that conditions which refer to these stabilities have to read the values from the agenda. The agenda consists of a sequence of excipient, stability pairs, so that to determine the stability of an excipient, it is necessary to read the agenda value immediately after that excipient. A test on the excipient for which MOST-STABLE-BINDER fires would therefore appear as follows.

```
<STABILITY> is after <BINDER> on STABILITY-AGENDA
<STABILITY> is less-than 42
```

The effect of the new rules

The above account has described how new rules are created. Now the effect of these rules is described. Considered as a set, the rules will fire for the same objects as before. (In the case of TFS, the objects are excipients). However, they will fire first for those objects which satisfy the induced conditions, which were chosen in such a way as to select the preferred objects. Where the conclusion of the original rule is successive or overwriting (section 5.3.2), the effect will be that the preferred objects are written first, for example, to an agenda. Where the original rule is self-disabling, as for example MOST-STABLE-BINDER is, the effect will be that only one of the rule-set $R \cup R_i$ will fire, and will do so for a preferred object.

6.4.1 The use of application-specific knowledge

This section examines the manner in which **inductive_split_rule** had to be tailored to the TFS application, and the implications for the general applicability of the technique.

The previous section showed that creating attribute vectors for use by ID3, and then translating ID3's output back into PFES rule conditions, was a complex process, requiring detailed knowledge of the manner in which TFS was implemented. However, it should be noted that this complexity was a direct consequence of the somewhat inconsistent manner in which attributes are represented in PFES; if the application had been developed with automated refinement in mind, then it is likely that attributes would have been handled more consistently. The conclusions from this experience are that PFES allows the programmer considerable freedom in knowledge representation, and that inconsistent or idiosyncratic programming styles will necessarily require the use of application-specific knowledge to allow an automated refinement tool to communicate with the application.

6.4.2 An example of the use of inductive_split_rule

Finally, an example is given of the application of the operator **inductive_split_rule** to TFS. In one case, TFS recommends the filler lactose, and the oracle recommends the filler magnesium carbonate. One of the rules responsible for choosing the filler is GET-ANY-FILLER, so **inductive_split_rule** is applied to this rule.

```
Rule GET-ANY-FILLER (Priority 0)
IF    on-agenda STABILITY-AGENDA <FILLER>
THEN
      set the value of FILLER in FORMULATION to be <FILLER>
UNLESS
      FILLER has value <ANY> in FORMULATION
```

When ID3 is run on the resulting attribute-vectors, it turns out that there exist two equally informative attributes, so that ID3 generates one of two trees, selected at random. The conditions which predict negative examples are either

```
YP-FAST of FILLER = <YP-FAST>
<YP-FAST> is less-than 451.3
```

or

```
CATEGORY of FILLER = <CATEGORY>

<CATEGORY> is less-than 1.5
```

When the first condition is generated, the following new rule is created with higher priority than GET-ANY-FILLER.

```
Rule RULE1 (Priority 1)
IF    on-agenda stability-agenda <FILLER>
 AND  YP-FAST of FILLER = <YP-FAST>
 AND  <YP-FAST> is less-than 451.3
THEN
      set the value of FILLER in FORMULATION to be <FILLER>
UNLESS
      FILLER has value <ANY> in FORMULATION
```

6.5 Using traces for refinement filtering

This section describes the second application of induction: refinement filtering. More effective refinement filtering is desirable, because it allows incorrect refinements to be removed without implementing and testing them. This is particularly important in the case of slow applications like TFS, where testing a KB is a time-consuming process.

The refinements generated by KRUST are derived from a single example of the system's incorrect conclusion and its proof, together with the expert's conclusion. Therefore, if KRUST also considers other proofs that are similar to the faulty proof yet lack the fault, it may obtain new information useful to the refinement process. The particular application considered here is to filter out refinements that are unlikely to fix the fault. A trace comparator is described which takes a pair of traces, and compares the firing behaviour for each of the rules for which KRUST proposes a repair. The procedure for filtering is as follows.

1. Let R be the set of rules which KRUST is refining.

2. Select sets of examples F and C, where examples F exhibit the fault (that exhibited by the current training example) and C do not.

3. Run the comparator for each rule $r_k \in R$ and for each pair of examples $(f_i, c_j) \in F \times C$. We define the comparator function $\text{diff}(r_k, f_i, c_j)$ to be 1 if the firing behaviour for rule r_k differs for examples f_i and c_j, 0 otherwise.

4. Then we say that the behaviour of rule r_k is relevant to the fault iff $\exists\ j$ such that $\forall i\ \text{diff}(r_k, f_i, c_j) = 1$.

Note that the appearance of a fault in one example and not in another may arise in two ways:

- it could be that a certain rule r fires in one case and not in the other, or
- it could be that r's firing behaviour is the same in both cases, but that this behaviour is faulty in the first case but correct in the second case.

The comparator will detect the difference in the first case, but not in the second. Hence we cannot choose as a criterion of relevance that the behaviour of r should differ for *all* faulty/non-faulty pairs.

6.5.1 Experimental results for refinement filtering

This section describes a simple experiment which demonstrates the effectiveness of refinement filtering. The procedure adopted was as follows:

1. Apply KRUST to a number of different refinement examples.

2. For each example, record which refinements were generated, and which was selected by KRUST as being the best.

3. For each example, apply the refinement filter. The filter will be regarded as useful if it rejects some unsuccessful refinements but does not reject the successful one.

The experiment was performed for three refinement examples, each demonstrating a different fault in TFS-1A. (The technique has not been applied to TFS-1B). A more detailed description of these faults and how they were fixed will be found in section 8.5.1.

For the purposes of the present discussion, it is sufficient to note that TFS-1A exhibits three different types of fault, each characterised by the particular attribute which is incorrect in TFS-1A's output.

Fault 1: Wrong binder weight.

Fault 2: Wrong filler.

Fault 3: Wrong target-tablet-weight.

For fault 3, KRUST generated refinements to a single rule only, so the algorithm was not required. The results of the experiment for the other two faults are summarised in tables 6.1 and 6.2. The second column in each of these tables indicates which rules were involved in the chaining process that lead to the faulty conclusion, and the third column shows which rules the trace comparator identified as potentially relevant to the fault. The tables show that, for both faults, the technique could be used to filter out some refinements that were indeed unrelated to the faulty conclusion, while not rejecting any that *were* relevant. The apparently poor behaviour for fault 1, where the technique highlighted only one of a possible three irrelevant rules, may be explained as follows. Fault 1 is a rarely-occurring fault, and all the examples of this fault happen to share certain other attributes: viz., they use no surfactant, and the drugs involved are soluble. These attributes are reflected in the firing behaviour of the rules related to these attributes (**Insoluble-Drug-Rule** and **Initial-Surfactant-Level**), so that the comparison algorithm also identified these rules as potentially relevant.

Rule	Involved in faulty conclusion	Trace comparator indicates relevance
Get-Soluble-Filler		
Insoluble-Drug-Rule		√
Get-Insoluble-Filler	√	√
Remove-Excessive-Fillers	√	√
Initial-Surfactant-Level		√

Table 6.1: Trace comparator applied to fault 1: Wrong filler

6.5.2 Effectiveness of refinement filtering

The experiment just described shows that for one group of examples, trace comparison can be used to reduce the number of refinements which have to be implemented and

Rule	Involved in faulty conclusion	Trace comparator indicates relevance
Default-Binder-Level	√	√
Update-Formulation	√	√
High-Dose-Binder-Level		√
Try-Dose-Again		
Find-Stable-Surfactant		
Default-Surfactant		

Table 6.2: Trace comparator applied to fault 2: Wrong binder level

tested. Further experiments would be necessary to show convincingly that the technique is both reliable and effective. However, I did not proceed to further test or develop the technique, because my work on the evaluation of KRUST, described in chapter 8, showed that KRUST did not generate large numbers of refinements when applied to TFS, so that an improvement in refinement filtering was not as important a goal as I had anticipated. Nonetheless, I believe the technique described in this section is interesting, since it illustrates an alternative application of induction, and that it would be useful in situations where KRUST did generate an excessive number of refinements.

6.6 Summary

This chapter has presented a common framework for the construction of inductive operators, and described four such operators. All these operators work by identifying features distinguishing traces which exhibit a certain faulty behaviour from traces that do not. These operators make more effective use of the training examples than KRUST's original operators, which learn only from the single refinement example. Moreover, they allow KRUST to learn new rule conditions for the first time, thus allowing it to fix faults which it was previously unable to fix.

Secondly, the same framework has been used to provide a mechanism for refinement filtering. Here, the particular feature selected to distinguish faulty from non-faulty traces is the firing behaviour of particular rules; it has been shown that it is possible to deduce from differences in firing behaviour whether a particular rule may be responsible for a given fault.

Chapter 7

Clustering

This chapter describes the ways in which KRUST uses clustering techniques to select examples for use at various stages of the refinement process. It is structured as follows.

- Background and motivation (section 7.1).
- How clustering could be used at various stages of the knowledge refinement process (section 7.2).
- Mechanisms for clustering (section 7.3).
- How a hierarchical clustering algorithm can be applied to TFS examples (section 7.4).
- A simpler clustering technique for TFS examples (section 7.5).

The chapter shows that a hierarchical clustering algorithm can be useful in some situations, but that a simpler technique is sufficient for the needs of KRUST when applied to TFS.

7.1 Motivation

Clustering is a process of grouping objects into classes of similar objects. It requires a measure of similarity to be defined between any two objects; classes can then be defined as collections of objects whose intraclass similarity is high and interclass similarity is low (Michalski & Stepp 1990).

The reason for considering clustering in the context of knowledge refinement is that it might provide KRUST with an intelligent mechanism for selecting examples. At various

stages of refinement, KRUST needs to select an example, or a set of examples, as one of the inputs to a procedure. For instance, a refinement example must be selected from the training set before KRUST can be run at all. Later on, sets of examples must be chosen for use by the inductive operators and for the final selection process. For those parts of the refinement procedure which are particularly time-consuming, it may be desirable to save time by using fewer than the full number of examples available. For instance, a single run of TFS can take between two and ten minutes. Consequently, it is desirable to reduce, if possible, the number of examples used by KRUST in the judging phase, where the number of TFS runs required is the product of the number of judging examples and the number of surviving refined KBs.

The simplest way of reducing the size of an example set is to select a random subset of examples, but it would be preferable to make a more intelligent selection. This could be done by first clustering the examples according to some definition of similarity, and then making a selection from the clusters. Depending on the purposes for which the examples were to be used, examples could be selected evenly from all clusters, or else from those clusters deemed particularly relevant.

In the context of knowledge refinement, there are several situations, listed below, for which the most useful clustering would be one in which examples belong to the same cluster if and only if they suffer from the same fault or faults, where a fault is defined to be a single error in the rule-base. Since the errors in the rule-base are initially unknown, however, the best that can be done is to use sources of information such as the system inputs and output to construct an approximation to this ideal clustering.

In the case of TFS-1A, the clustering task is relatively trivial and can be performed either by hand or mechanically, just by considering which fields in the system's formulation for each example are incorrect. Figure 7.1 shows typical differences; in each pair, the first item is the system output and the second that of the oracle. There are three types of error:

- Wrong quantity of binder (examples 1,2,5,6,29).
- Wrong filler (examples 5, 33).
- Numerous errors in specification (examples 206, 207).

Example	Attribute	System Value	Expert Value
1	Binder	Gelatin 0.041	Gelatin 0.021
2	Binder	Gelatin 0.041	Gelatin 0.021
5	Filler	Calcium Phosphate 0.573	Calcium Carbonate 0.585
	Binder	Gelatin 0.041	Gelatin 0.021
6	Binder	Gelatin 0.041	Gelatin 0.021
29	Binder	Pvp 0.039	Pvp 0.020
33	Filler	Calcium Phosphate 0.554	Magnesium Carbonate 0.565
	Binder	Pvp 0.039	Pvp 0.020
206	Target-Tablet-Weight	400	450
	Drug-Concentration	9/10	4/5
	Filler-Concentration	0.0	0.1
	Current-Strategy	Strategy-L	Strategy-D
	...		
207	Target-Tablet-Weight	400	450
	Drug-Concentration	9/10	4/5
	Filler-Concentration	0.0	0.1
	Current-Strategy	Strategy-L	Strategy-A
	...		

Figure 7.1: Differences between the system and oracle outputs for TFS-1A

However, to generate appropriate clusters in situations where many examples suffer from multiple faults, further inputs to the clustering process could be considered, such as the example traces, or even the outputs from KRUST's rule classification and refinement modules. Whether the extra computation is justified depends on the potential gains and the need for accuracy in clustering. This point is discussed later on, with reference to inductive operators (section 7.4).

7.2 Purposes of clustering

Clustering is applicable at a number of points in KRUST's refinement procedures.

1. **To select refinement examples for KRUST.**

   In TFS-1A, an automated clustering algorithm is not strictly necessary for the selection of training examples. This is because it is possible to distinguish individual fault symptoms, and determine which examples are suffering from a single fault, and which from multiple faults. For example, figure 7.1 shows that many examples have approximately twice the correct quantity of binder, but are otherwise correct. This suggests that example 5, which has incorrect filler as well as too much binder, is

suffering from two separate and unrelated faults. However, examples 206 and 207 (and many others) have faults lying in the same specification fields. This suggests that these examples are suffering from a single fault with multiple consequences. Examples suffering from single faults are likely to be easier to fix than ones suffering from multiple faults, and it is possible that refinements which fix all the single faults may together fix the multiple fault examples as well. Therefore, a sensible initial procedure for selecting refinement examples is to choose a sequence of examples where each suffers from a single, different, fault.

In general, however, it may be hard to distinguish examples suffering from single faults from those suffering from multiple faults. In this case, the decision of whether to devote resources to a clustering algorithm that can identify single-fault examples will depend on how much harder it is for KRUST to generate appropriate refinements for multiple-fault examples than for the single faults. In any case, it may still be preferable to deal with the single fault examples first.

2. **To select appropriate test examples for judging refined KBs.**
   Given a clustering close to the ideal one, that is, one which corresponds to the underlying faults in the KB, it would be expected that a correct refinement would fix most or all of the examples in the refinement example's cluster, and few others. The time taken by the judgement phase could be reduced by selecting examples mainly from this cluster, and from the cluster holding correctly-solved examples, but using only a few examples suffering from unrelated faults. For this purpose, it would not matter if the clustering were not 100% correct.

3. **To select appropriate examples for induction.** Refinements involving the induction of new facts or rule conditions require multiple examples. The inductive operators themselves are able to select appropriate positive and negative examples (chapter 6), but it may be desirable to reduce the number of examples which are available to them in order to speed up the process. This could be done by clustering the examples into fault types, and selecting examples for the inductive operators evenly from these clusters.

7.3 Mechanisms for clustering

The previous sections have explained how an example clustering mechanism could be used by KRUST. Here, various approaches to clustering are outlined.

All clustering methods require that a distance function be defined between all pairs of items in the set to be clustered. Clustering methods can be divided into the simpler *non-hierarchical* and the more complex *hierarchical* (Rasumssen 1992). Non-hierarchical methods partition a set into a given number of subsets; hierarchical methods produce a nested data set in which pairs of items or clusters are successively linked until every item in the data-set is connected. Hierarchical methods are usually *agglomerative*, or bottom-up, building small clusters and gradually linking them. The alternative, *divisive* methods, are rarely used.

All the hierarchical agglomerative clustering methods can be described by the following algorithm:

```
While more than one cluster remains
    Identify and combine the two closest items (clusters or points)
```

This algorithm requires that the distance between two *clusters* be defined in some way, in terms of the distances between items in the respective clusters. Different versions of the algorithm are distinguished by the way in which this distance is defined. There are three commonly used definitions.

The single link method. Cluster distance is defined as the minimal distance between an item in one cluster and one in the other.

The complete link method. Cluster distance is defined as the maximal distance between an item in one cluster and one in the other.

The group average link method. The average of all pairwise links is used.

The complete link method tends to form small, tightly bound clusters (Rasumssen 1992). For this reason, when a hierarchical clustering algorithm was implemented for TFS ex-

amples, the complete link method was chosen (section 7.4). In addition, a simpler non-hierarchical method was also implemented (section 7.5.)

7.4 Applying clustering to knowledge refinement

This section describes the application of a hierarchical clustering method to TFS examples. Before a clustering algorithm can be applied to a set of objects, a numerical measure of distance between objects has to be defined. In the case of TFS, it turned out to be easier to define first a measure of similarity, and then define distance to be the inverse of similarity. For TFS examples, the information from which this measure must be calculated lies in the system and oracle values for all the attributes appearing in the formulation. Given a pair of examples, for each attribute i there are four values to be considered.

	Expert	System
Example A	$E_{A,i}$	$S_{A,i}$
Example B	$E_{B,i}$	$S_{B,i}$

The simplest measure of similarity is to count the number of fields in which the examples have the same correctness: for each attribute, score one point if the values are either both correct or both wrong, and sum these values over all the attributes. This measure is expressed formally by the definition 7.1.

$$\text{Similarity}_{A,B} = \Sigma_i(E_{A,i} = S_{A,i}) = (E_{B,i} = S_{B,i}) \tag{7.1}$$

However, there are other features which may also indicate that examples share a fault. For example, the oracle value may be the same for a group of examples, perhaps because a rule that would have generated that value failed to fire in all cases. Alternatively, if the system value is the same for a group of examples, a rule generating that value may have fired incorrectly. For example, section 5.5.1 described a fault which caused TFS to recommend the filler Calcium Phosphate on several occasions when it should not have done. The symptom of this fault was that the system value of filler was Calcium Phosphate; the oracle's value for filler varied. The following similarity measure extends definition 7.1 by adding terms which recognise if the system or oracle values for some field are the same for

a pair of examples.

$$\text{Similarity}_{A,B} = \Sigma_i((E_{A,i} = S_{A,i}) = (E_{B,i} = S_{B,i})) + (E_{A,i} = E_{B,i}) + (S_{A,i} = S_{B,i}) \quad (7.2)$$

This measure is superior to definition 7.1 in that it could, for example, distinguish examples of the Calcium Phosphate fault just described from other examples also affecting the filler.

A hierarchical clustering algorithm was implemented using the complete link method, and the similarity measure 7.2. This algorithm was run on the TFS-1A examples. Figure 7.2 shows a typical hierarchical structure, or dendrogram, generated by the algorithm for a subset of the examples. Various different partitions of the example set can be derived from the dendrogram, depending on the distance threshold at which the process of combining clusters is stopped. Given a sufficiently high threshold, the partition will consist of a single cluster holding all the examples; given a sufficiently low threshold, each example will lie in a cluster of its own. The grey bar in figure 7.2 shows the range of thresholds that will lead to a partition where each cluster corresponds to a particular fault, or combination of faults.

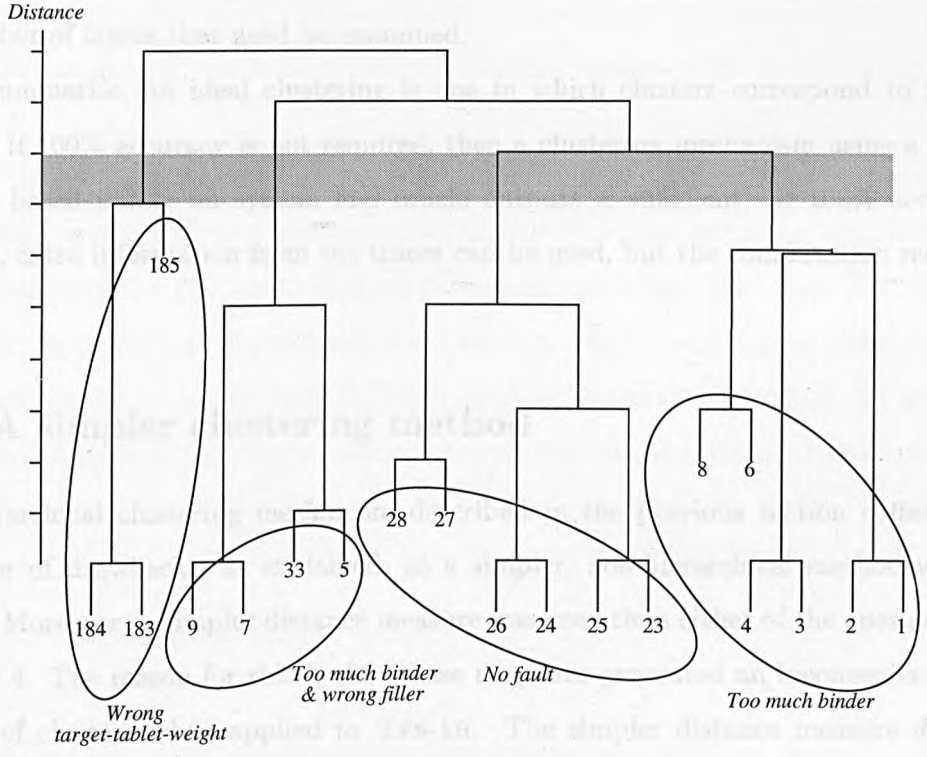

Figure 7.2: Dendrogram for some TFS-1a examples

It will be seen that the hierarchical clustering method has the disadvantage of providing more structure than is needed for selecting examples; moreover, there is no obvious way of choosing a threshold value that will generate an appropriate example partition from the tree. One approach might be to use, say, the ratio of average inter-cluster distance and average intra-cluster distance. However, for selecting refinement and judging examples for TFS, the hierarchic method just described was in the end not employed, because the simpler non-hierarchic method defined below (section 7.5) proved sufficient. Consequently, methods for selecting an appropriate threshold for partitioning the hierarchical clustering were not investigated further.

It will be noted that the inductive operators described in chapter 6 also perform a kind of clustering, since they require sets of positive and negatives examples of the fault they are trying to fix. In their case, greater precision is required, since a single incorrectly classified example could lead to an incorrect result, which is not the case when using clustering to select refinement or judging examples. Consequently, these inductive operators use further information about rule firings derived from the trace, in addition to the system and oracle outputs. However, the computation required for this is minimised by using the system and oracle outputs first to select candidate positive and negative examples, thus reducing the number of traces that need be examined.

To summarise, an ideal clustering is one in which clusters correspond to faults in the KB. If 100% accuracy is not required, then a clustering mechanism using a distance function based purely on system and oracle outputs is sufficient. If 100% accuracy is required, extra information from the traces can be used, but the computation required is greater.

7.5 A simpler clustering method

The hierarchical clustering mechanism described in the previous section suffered from a number of drawbacks, as explained, so a simpler, non-hierarchical method was used instead. Moreover, a simpler distance measure was used than either of the ones defined in section 7.4. The reason for this is that these measures generated an inconveniently large number of clusters when applied to TFS-1B. The simpler distance measure $d(E_1, E_2)$ between examples E_1 and E_2 is defined as follows. $d(E_1, E_2) = 0$ if the errors in the

system output for the two examples lie in exactly the same fields, 1 otherwise. Thus, if two examples both have incorrect filler, but are correct in all other aspects, the distance between them will be 0.

This definition needs to be extended slightly to take into account fields which include both an excipient and its quantity. For such fields three cases are considered:

1. both values are correct;
2. the excipient is correct, but the quantity is wrong;
3. the excipient is wrong.

The case where the excipient is wrong but the quantity is correct is not considered, since the quantity depends on the choice of excipient; if the excipient is wrong, then it is only by chance that the quantity could be correct. For excipient/value fields, examples are regarded as suffering from the same fault if and only if they both lie in the same error-class for that field. For instance, an example exhibiting a correct binder but wrong binder quantity does not lie in the same error class as an example exhibiting an incorrect binder. $d(E_1, E_2)$ is now defined to be 0 if and only if E_1 and E_2 have the same type of fault for each output field. The effect of this distance measure is to associate together examples which differ in exactly the same attributes, without regard to the actual values of these attributes.

Now that a distance measure has been defined, it is possible to define a clustering mechanism. Non-hierarchical clustering can be carried out by assigning examples to cells in an n-dimensional array, where each cell represents a particular fault type, or combination of fault types, as follows. An n-dimensional array is created, where each dimension of the array corresponds to an attribute in TFS's output. For example, if just two attributes, Binder and Target-Tablet-Weight, are considered, the following array will be created.

	Binder
Target-Tablet-Weight	

For attributes referring to excipients, such as Binder, the length of the axis is 3, corresponding to correct excipient and quantity, wrong quantity, and wrong excipient. For

other attributes, the length of the axis is 2, corresponding to correct value, and wrong value.

		Binder		
		Correct	Wrong quantity	Wrong binder
Target-Tablet-Weight	Correct			
	Wrong			

Each cell of the array will hold examples whose errors correspond to the co-ordinates of that cell. Here some examples have been placed in the array, to illustrate how this works.

		Binder		
		Correct	Wrong quantity	Wrong binder
Target-Tablet-Weight	Correct	1		3,4
	Wrong		2	

Example 1 in the above array is correct in all respects, example 2 has the wrong target-tablet-weight and the wrong quantity of binder, and examples 3 and 4 have the correct target-tablet-weight but the wrong binder.

The array is used at two stages during the refinement process: to select refinement examples, and to select judging examples. A procedure is required for selecting a subset of the training examples for use in judging, because judging requires the running of TFS, which is very time-consuming. Otherwise, the entire training set could be used.

The procedure for selecting refinement examples is as follows. Let those cells which are occupied, and which do not hold correctly solved examples, be C_i, $1 \leq i \leq p$. The cell which holds correctly solved examples is omitted, since these are of no use as refinement examples.

```
for i = 1 to p
    Select an example e at random from cell C_i
    Run KRUST on example e
end
```

This selects a single refinement example in turn from each cell containing at least one faulty example.

The algorithm for selecting a subset of examples for judging aims to select an equal number of examples from each occupied cell in the array, to the extent that the number

of examples in each cell permit this. Let the occupied cells be C_i, $1 \leq i \leq q$. Then the selection algorithm to select n examples is as follows.

```
Let example-set = {}
While | example-set | < n
    for i = 1 to q
        If cell C_i contains at least one example, then
            remove one example at random from cell C_i
            and add it to example-set
        If | example-set | = n then exit
    end
end
```

This picks examples one at a time from each occupied cell, starting again at the beginning as necessary, until the required number of examples have been selected.

7.6 Summary

This chapter has shown how example clustering can be used by KRUST, and presented two clustering methods. The next chapter describes the evaluation of KRUST when applied to TFS-1A and TFS-1B; this evaluation uses the simpler clustering method described in section 7.5.

Chapter 8

Evaluation

Traditionally, developers of refinement systems have adopted evaluation techniques borrowed from the machine learning community. Typical techniques include testing whether the tool can fix artificial corruptions in a KB, and running the tool on a large number of examples in order to produce learning curves. Work described elsewhere demonstrates KRUST's ability to refine small or artificial KBSs, or KBSs with artificial corruptions (Craw & Hutton 1995, Palmer & Craw 1996). This work uses traditional evaluation methods to show that KRUST is competitive with other systems when applied to KBSs of this type.

However, traditional evaluation methods are less appropriate to the work described in this thesis. Instead, the thesis aims to show that KRUST is of practical use in software development, since it can fix real bugs which actually occur in an application. This chapter first uses the history of TFS to illustrate a typical software development cycle, and explains where knowledge refinement can be applied within that cycle (section 8.1). It then outlines traditional evaluation methods (section 8.2), and goes on to show how these methods can be adapted to make them more suitable for evaluating KRUST's effectiveness as a practical tool (section 8.3). Finally, it describes an experimental design for evaluating KRUST's effectiveness in refining TFS (section 8.4) and presents results for TFS-1A and TFS-1B (sections 8.5 and 8.6).

8.1 The role of refinement in software development

Figure 8.1 shows the place of TFS-1A, TFS-1B and TFS-2 within the development of TFS. The first version, TFS-1A was created from scratch by traditional means of knowledge

elicitation, such as interviewing experts, and protocol analysis. TFS-1B was created by a process of debugging; that is, making small changes to fix faults in TFS-1A. TFS-2 was created as a result of more substantial modifications which were required by a change in the specification of the formulation process. TFS-3 was created as a result of an even more substantial redesign of the system, when the requirements of TFS changed.

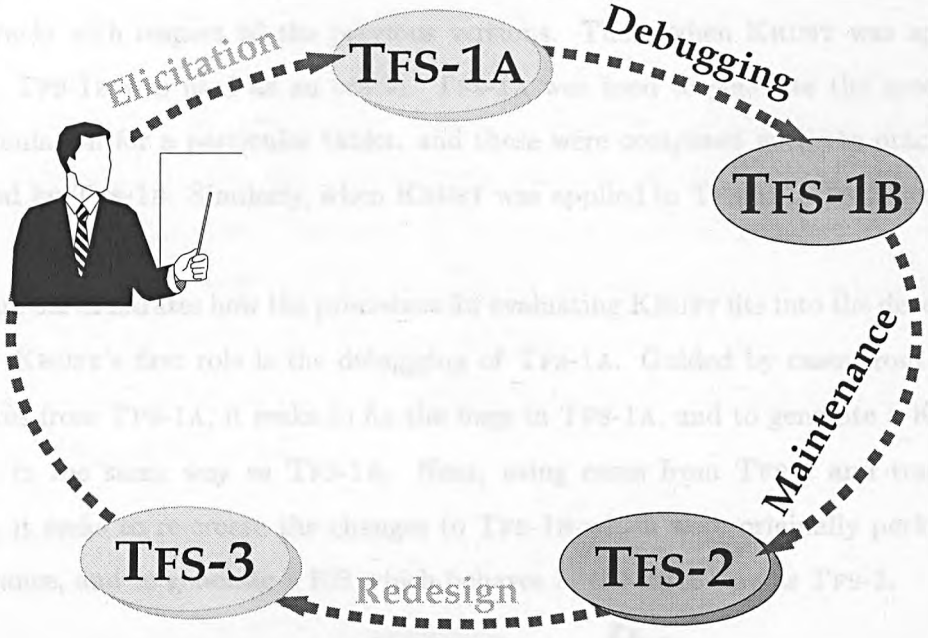

Figure 8.1: Stages in the development of TFS

Traditionally, AI techniques can be used at two stages in the above typical software development cycle. Machine Learning can be used to assist in knowledge elicitation, and knowledge refinement can be used to assist with debugging. However, this thesis aims to show that the role of knowledge refinement can be extended by applying it also in the maintenance stage.

The most straightforward way to evaluate KRUST's applicability to TFS would have been during TFS's development. KRUST might have been used to assist with the debugging of TFS-1A and with the maintenance work on TFS-1B. However, this was not possible, since TFS-2 was already complete before my project started. However, the existence of all three versions of TFS enabled the evaluation to be carried out retrospectively, and made it possible to judge how closely KRUST replicated the manual development process. The details of how this was done are described in the next section.

8.1.1 The use of TFS as an oracle

Discussions of faulty behaviour of a KBS have described the fault in terms of differences between the behaviour of the system and the behaviour recommended by an oracle. Typically the oracle is a human expert. However, the fact that I had access to a number of different versions of TFS allowed a different approach: to regard each version of TFS as an oracle with respect to the previous versions. Thus, when KRUST was applied to TFS-1A, TFS-1B was used as an oracle. TFS-1A was used to generate the specification and formulation for a particular tablet, and these were compared with the oracle values generated by TFS-1B. Similarly, when KRUST was applied to TFS-1B, TFS-2 became the oracle.

Figure 8.2 illustrates how the procedure for evaluating KRUST fits into the development of TFS. KRUST's first role is the debugging of TFS-1A. Guided by cases from TFS-1B, and traces from TFS-1A, it seeks to fix the bugs in TFS-1A, and to generate a KB which behaves in the same way as TFS-1B. Next, using cases from TFS-2 and traces from TFS-1B, it seeks to re-create the changes to TFS-1B which were originally performed as maintenance, and to generate a KB which behaves in the same way as TFS-2.

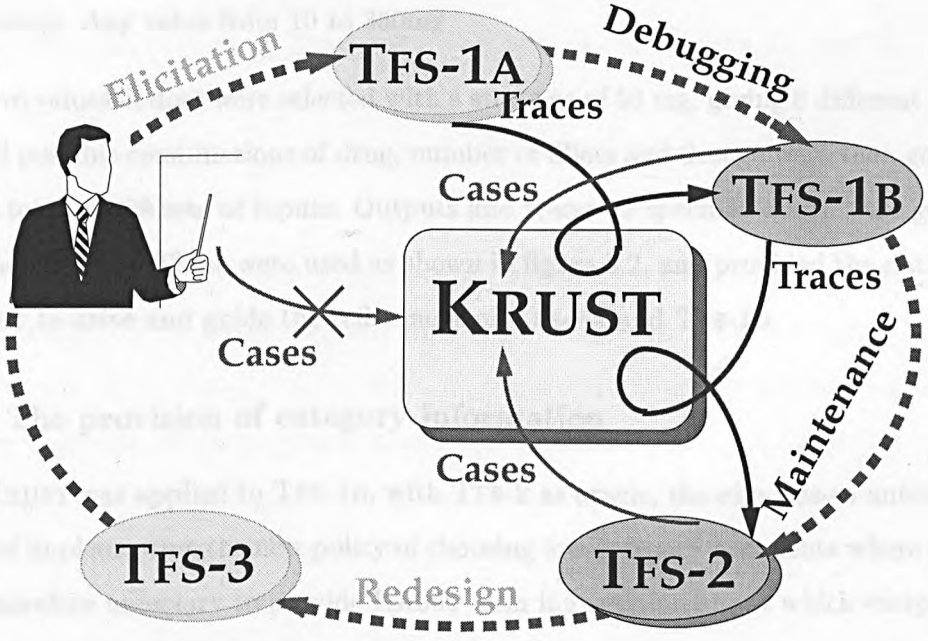

Figure 8.2: How KRUST was applied to different versions of TFS

Since I was provided with copies of TFS-1A and TFS-1B, it would have been possible, when refining TFS-1A, to run the appropriate version of TFS whenever the system or

oracle outputs for a particular example were required by KRUST. However, this would have been very slow and inefficient, since it would have lead to the same examples being run many times. Moreover, outputs from TFS-2 could not be acquired in this way, since I was not provided with a copy of this version, but had to visit Zeneca in order to run it. For these reasons, I chose instead to generate results from all versions of TFS for a large number of examples, and store them for later use.

KRUST requires outputs from both the system being refined and from the oracle, but it requires traces from the system only. Indeed, it would be inappropriate to allow KRUST access to the oracle traces, since these convey information about the refined KB which KRUST is trying to generate. Normally, of course, the oracle is an expert, not a piece of software, so that oracle traces will not be available. Consequently, outputs were generated for TFS-1A, TFS-1B and TFS-2, and traces for TFS-1A and TFS-1B.

The sets of inputs for the examples was selected so as to cover the input space evenly. The range of inputs was as follows:

Drug: A, E, F, G, H, I, K, M, N, P, R, S, or T[1]

Number of fillers: 1 or 2

Dosage: Any value from 10 to 360mg

Successive values of dose were selected with a step size of 50 mg, giving 8 different values of dose. All possible combinations of drug, number of fillers and dosage were then generated, giving a total of 208 sets of inputs. Outputs and traces as specified above were generated for all these inputs. These were used as shown in figure 8.2, and provided the data needed by KRUST to drive and guide the refinement of TFS-1A and TFS-1B.

8.1.2 The provision of category information

When KRUST was applied to TFS-1B, with TFS-2 as oracle, the aim was to automate the process of implementing the new policy of choosing low-category excipients where possible. It was therefore necessary to provide KRUST with information about which excipients lay in which categories. This was not unreasonable, since the developers who performed the task manually were provided with the same information.

[1]The actual names of the drugs were replaced by letters, for reasons of confidentiality

The category information was added to the TFS-1B database, and was presented in the same way as the existing chemical data. For example, the fact that lactose is a category one excipient was represented by the fact

```
CATEGORY of LACTOSE is 1
```

8.2 Traditional evaluation methods for learning algorithms

This section summarises traditional evaluation methods. The summary will provide the background for the next sections, which describe how traditional methods can be modified to make them more applicable to current needs, where a refinement tool is to be evaluated for the tasks of software debugging and maintenance.

A typical learning algorithm takes as input a set of pre-classified training examples L, and generates as output a procedure or structure R which may be used for classifying further examples. R may be, for example, a decision tree, rule set, or trained neural net.

The simplest approach to evaluating such an algorithm A, given a set E of classified examples, is as follows:

Iteration i

Randomly partition E into learning and testing sets L_i and T_i

Apply A to L_i to learn R_i.

Evaluate the accuracy of R_i on T_i; that is, determine what proportion of T_i are classified correctly by R_i.

A possible enhancement to this is, for each division of E into L_i and T_i, to partition the learning examples L into equal subsets $L_{i,1}, L_{i,2}, \ldots L_{i,n_i}$ and apply the algorithm in turn to the sets $L_{i,1}, L_{i,1} \cup L_{i,2}, \ldots, L_{i,1} \cup L_{i,2} \cup \ldots L_{i,n_i}$, evaluating the accuracy for each, thus obtaining a typical learning curve as shown in figure 8.4.

This approach can be most efficiently implemented for incremental learning algorithms for which the results of processing set L_1 followed by set L_2 are the same as the results of processing the combined set $L_1 \cup L_2$.

8.2.1 The non-incremental nature of KRUST

Given a learning set L_i consisting of examples $l_{i,1}, l_{i,2} \ldots l_{i,m_i}$, KRUST's approach is to select each $l_{i,j}$ in turn as a training example, and to use the complete example set for filtering

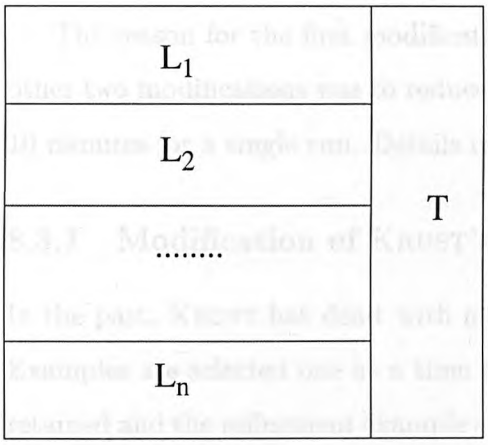

Figure 8.3: Incremental Learning

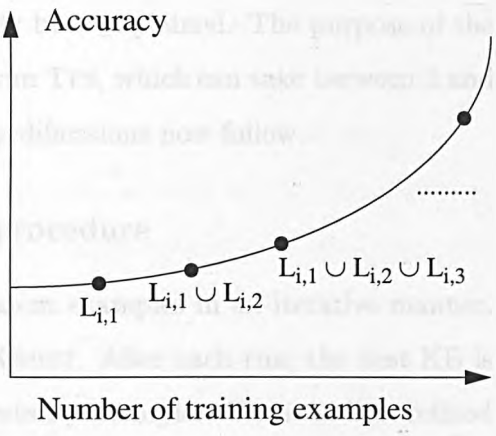

Figure 8.4: Associated learning curve

and induction. When KRUST processes L_1, only the examples in L_1 will be available for filtering and induction. However, when KRUST processes $L_1 \cup L_2$, the extra examples in L_2 will become available for use with the refinement examples from L_1. Therefore, for each augmentation of the example set, KRUST will have to start from scratch, rather than continuing from where it left off. It follows that KRUST is *not* an incremental learning algorithm. Hence the approach to testing depicted in figures 8.3 and 8.4 is not particularly efficient for KRUST, and may not be feasible for large or slow applications. This restriction is not unique to KRUST; it applies to any refinement tool. It would not apply to a tool that learned from a single example at a time, while making no use of any other examples, but I am not aware of any such refinement tool.

8.3 How evaluation methods can be adapted to the needs of KRUST

This section describes and justifies the ways in which the usual procedure for applying KRUST to a KBS were modified during KRUST's evaluation on TFS. The following modifications were made.

- The oracle's formulations were provided by versions of TFS, not by a human expert.
- KRUST did not iterate over a sequence of examples.
- Clustering was used to select refinement and judging examples.

The reason for the first modification has already been explained. The purpose of the other two modifications was to reduce the need to run TFS, which can take between 2 and 10 minutes for a single run. Details of these two modifications now follow.

8.3.1 Modification of KRUST's iterative procedure

In the past, KRUST has dealt with multiple refinement examples in an iterative manner. Examples are selected one at a time for input to KRUST. After each run, the best KB is retained and the refinement example added to the priority examples. Thus the best refined KB generated by each run is used as input to the next run, so that KRUST generates a series of KBs, each more accurate than the one before.

The disadvantage of this approach when applied to TFS arises because of KRUST's use of traces. Traces are required both for the refinement example, and for any examples used for induction; unfortunately, after the first iteration of KRUST has lead to the generation of a new KB, all the old traces become obsolete. Generating a complete set of new traces for all the training examples is possible but would consume much time and file-space. One possible compromise would be to generate a new trace for each refinement example, but to retain the old traces for induction. (figure 8.5). The result of this would be to save considerably on time and space, at the expense of an increasing loss in accuracy in induction as the iteration proceeded.

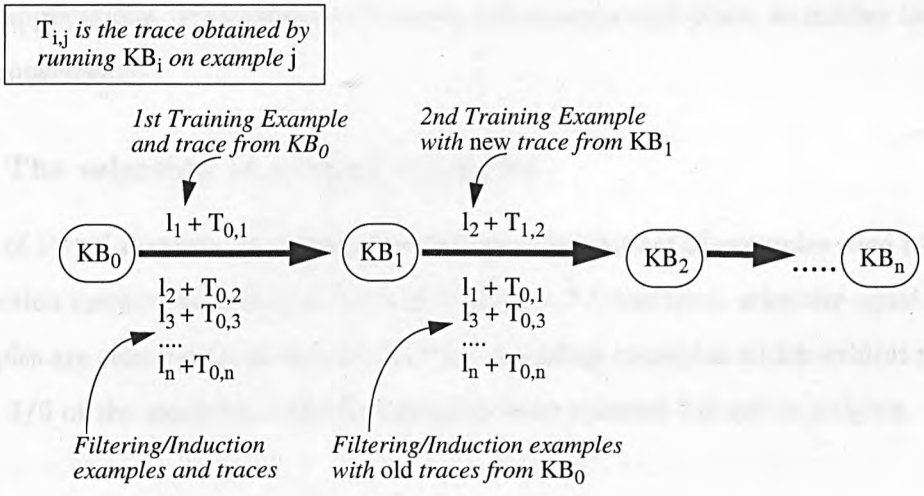

Figure 8.5: Applying KRUST iteratively to TFS

A better approach would be that shown in figure 8.6. Here KRUST is run separately for each refinement example $l_1, l_2, \ldots l_n$, starting in each case with the original KB, KB_0.

The result is a set of KBs, $KB_1, KB_2, \ldots KB_n$, where each refined KB, KB_i, is generated by applying refinement r_i to KB_0. Once this has been done, the refinements corresponding to each new KB can be combined into a single KB which incorporates all of them.

This approach has two potential problems. First, two or more of the refinements r_i might be incompatible. For example, one refinement might specialise a condition, and another might generalise the same condition. Secondly, it is possible that two refinements, while not being incompatible, might nonetheless interfere with one another, so that, while $r_i(KB_0)$ fixed example l_i, the combined refinement $r_j r_i(KB_0)$ did not. Currently, KRUST recognises the first of these problems. It constructs the combined refinement by applying the constituent refinements one at a time, and if it encounters a refinement which is incompatible with a previous refinement, it simply fails to apply it. Consequently, the presence either of incompatible or of interfering refinements may mean that the combined refinement fails to fix one or more of the refinement cases. The simplest way to deal with this situation would be to test the resulting KB on all the refinement cases, and to identify cases on which the KB fails. Further iterations of the procedure shown in figure 8.6 could then take place, starting from the refined KB just generated, and using the previously failing cases as refinement examples. In pathological cases, it might be necessary to revert to a purely serial procedure, as shown in figure 8.5.

In practice, when the technique of parallel refinement shown in figure 8.6 was applied to the TFS applications, no interference between refinements took place, so further interations proved unnecessary.

8.3.2 The selection of judging examples

Because of PFES' slowness, it is desirable to reduce the number of examples used in judging. The selection method described at the end of section 7.5 was used, whereby equal numbers of examples are selected from each fault class (including examples which exhibit no fault). In total, 1/6 of the available training examples were selected for use in judging.

8.4 Experimental design

The experimental design for evaluating KRUST followed the approach in section 8.2, but did not incorporate the technique of incremental evaluation. The example set was re-

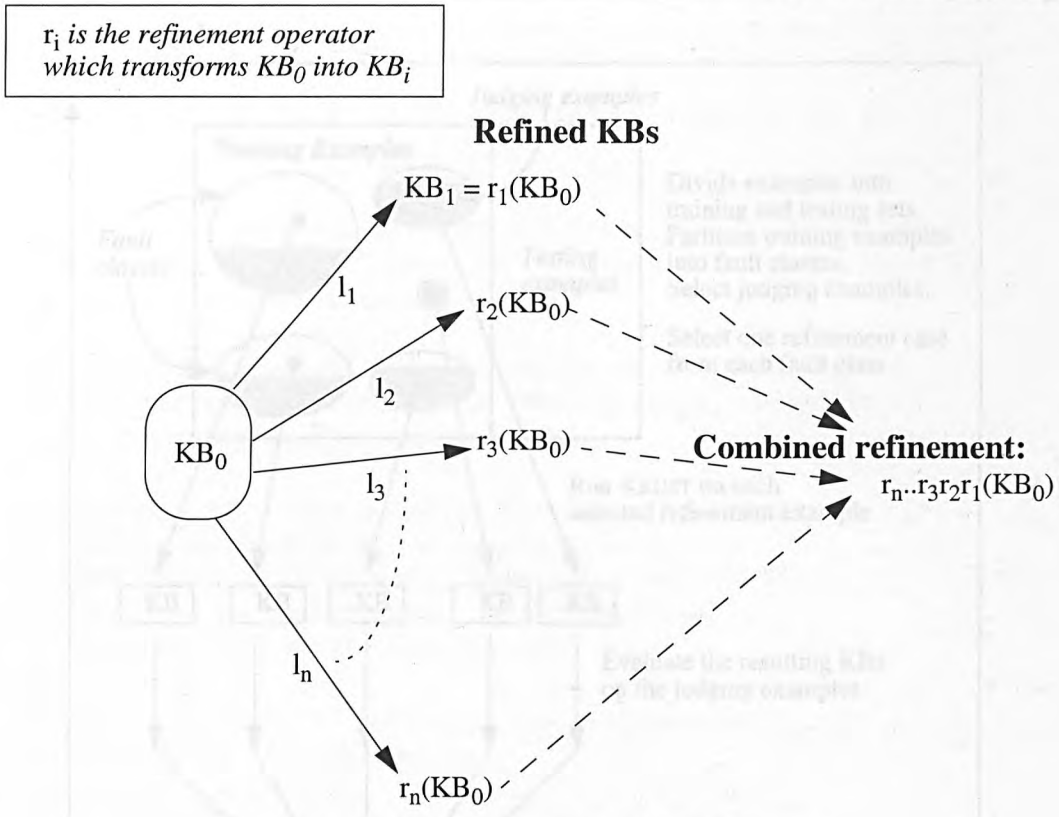

Figure 8.6: An alternative to iteration

peatedly partitioned into training and testing sets. After each partition, the training set was clustered according to fault type, using the simple non-hierarchical mechanism described in section 7.5. For the purposes of example selection only, clusters sharing the same independent incorrect attribute were merged. For example, in the case of TFS-1A, the partition included a cluster where the only fault lay in the target-tablet-weight, and a second cluster where faults lay in both target-tablet-weight and filler. For examples from either cluster, the independent attribute, which is used to drive refinement, is target-tablet-weight. Consequently, the two clusters were merged. After the merging process, one example was then selected in turn from each cluster, and used as a refinement example by KRUST. The best refined KB generated from each run was then evaluated on the testing set. Furthermore, once one example of each fault had been processed, the refinements associated with the best KB from each run were combined and implemented, as shown in figure 8.7. The resulting KB was then itself evaluated on the testing examples. This algorithm is described more formally in figure 8.8.

The following method was used for evaluating each KB on the testing set. For each

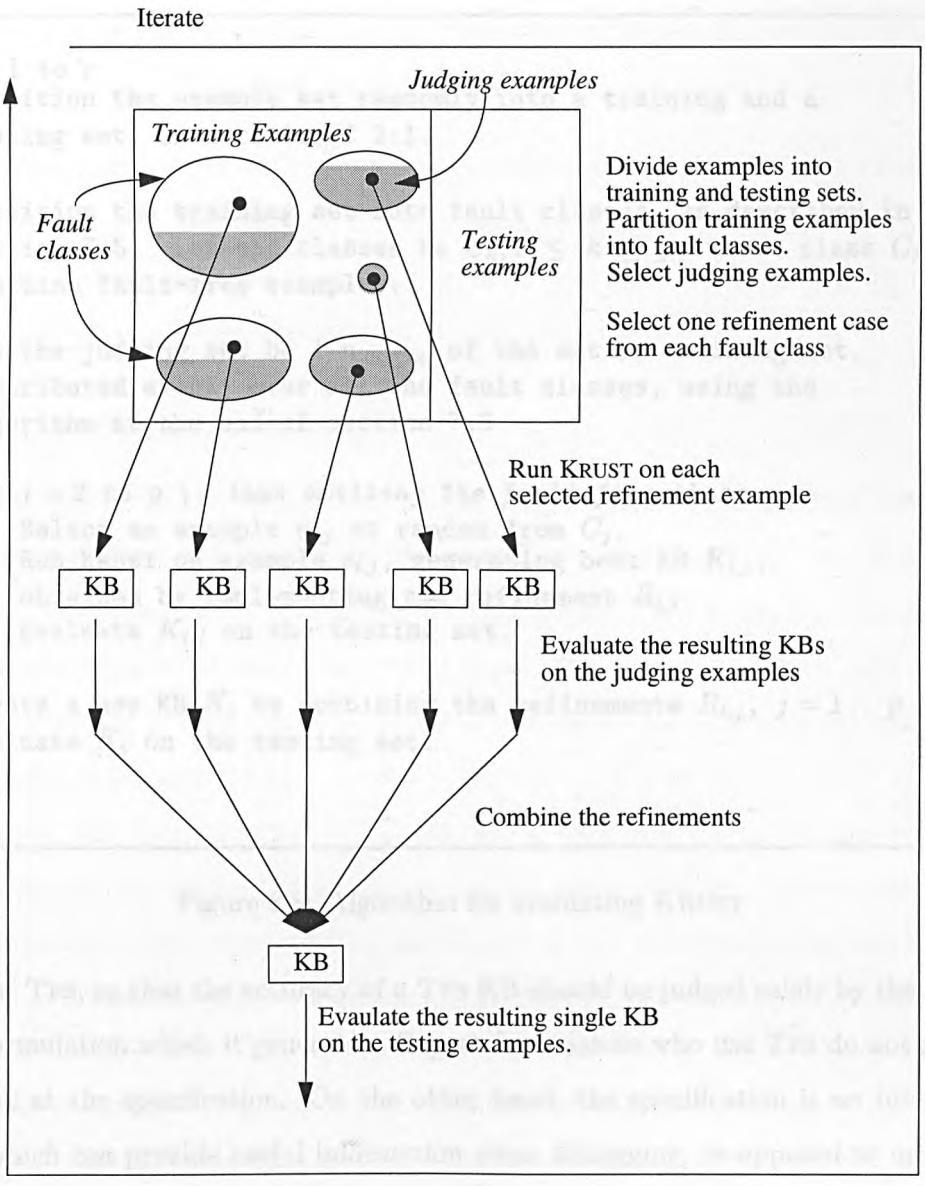

Figure 8.7: The evaluation of KRUST on an example set

example, the formulation generated by the KB was compared with the oracle's formulation, and the number of errors counted. Any field that was not correct in both excipient and quantity was regarded as one error. The error-rate for the KB was then calculated as

$$\frac{\text{total errors}}{\text{number of fields in formulation} \times \text{number of examples}}$$

Note that this measure makes use of the formulation only, whereas KRUST's filtering and judging modules use both formulation and specification. The reason for including only the formulation in the evaluation measure is that the formulation constitutes the useful

```
For i = 1 to r
    Partition the example set randomly into a training and a
    testing set, in a ratio of 2:1.

    Partition the training set into fault classes, as described in
    section 7.5.  Let the classes be C_k, 1 ≤ k ≤ p, where class C_1
    contains fault-free examples.

    Let the judging set be 1/n_judging of the entire training set,
    distributed evenly over all the fault classes, using the
    algorithm at the end of section 7.5

    For j = 2 to p ;; thus omitting the fault-free class
        Select an example e_{i,j} at random from C_j.
        Run KRUST on example e_{i,j}, generating best KB K_{i,j},
        obtained by implementing the refinement R_{i,j}.
        Evaluate K_{i,j} on the testing set.
    End
    Create a new KB K_i by combining the refinements R_{i,j}, j = 1...p
    Evaluate K_i on the testing set.
End
```

Figure 8.8: Algorithm for evaluating KRUST

output of TFS, so that the accuracy of a TFS KB should be judged solely by the accuracy of the formulation which it generates. Expert formulators who use TFS do not generally even look at the specification. On the other hand, the specification is an intermediate result, which can provide useful information when debugging, as opposed to using, TFS. It is therefore reasonable for KRUST to have access to both specifications and formulations when debugging TFS KBs.

8.5 Evaluation of Krust as applied to TFS-1A

The experiment depicted in figure 8.7 was run with 10 iterations. The partitioning procedure revealed the following fault types. Examples of these were shown in the previous chapter (figure 7.1).

- Wrong quantity of binder.
- Wrong filler.

- Numerous errors in specification and formulation. For each such example, the one independent attribute was target-tablet-weight.

After the merging process described in the previous section, three clusters remained, one for each fault type. In principle, a set of training examples might have been selected which contained no examples of a particular fault type. In practice, this never happened; examples of all three faults were present in each iteration.

8.5.1 Experimental results

When the experiment had been run, the single best refined KB generated for each example was examined. The most important observation was that given any one example of a particular fault type, KRUST was able to generate a refined KB which fixed the refinement example, and most other examples of that type. The manner in which each fault type was fixed is shown in table 8.1. The details of the repairs for each fault type are given in sections 5.5.4, 6.2.1 and 6.1.1.

Moreover, the domain expert confirmed that in each case the fix was the correct one, or an acceptable alternative. Furthermore, the KB which combines the refinements for all the different faults gives results which correspond almost exactly with the oracle.

There were just two respects in which the changes which KRUST recommended to TFS-1A did not correspond precisely to those introduced in TFS-1B. First, KRUST was unable to arrive at the precise value for the maximum level for calcium phosphate which was used in TFS-1B. This was because the available training examples did not sufficiently constrain the choice of maximum level. Secondly, KRUST was unable to learn the formula used by TFS-1B for calculating target-tablet-weight for high values of dose. This was because the new formula applied to only a very limited range of doses, and furthermore the new formula used a polynomial to calculate the target-tablet-weight, and then rounded the result to the nearest 5mg (this is described in more detail in section 6.1.1). Consequently, the data available were insufficient to learn the new polynomial. However, the alternative formula learned by KRUST generated the same results as the TFS-1B formula for all the available examples.

A statistical summary of the accuracy of the refined KBs is given in table 8.2. Each row of this table summarises the performance of KRUST on one example of each fault type.

Fault 1	Wrong binder weight.	RHS of default-binder-level changed from 0.04 to 0.02
Fault 2	Wrong filler	The missing value of max-level of calcium phosphate was added to the database. This caused calcium phosphate to be rejected when it exceeded the max level.
Fault 3	Wrong target-tablet-weight	A new rule was learned for calculating target-tablet-weight for high values of dose.

Table 8.1: Summary of how KRUST fixed the three faults

The successive columns represent the different fault types; each entry shows the error-rate for the best refined KB resulting from a refinement example of that fault type. The final column shows the error-rate of the KB which combines these refinements.

	Error rates for KB				
	Original KB	Best KB, when refinement case is taken from the given fault class			KB combining refinements
Run		Fault 1	Fault 2	Fault 3	
1	0.210	0.048	0.186	0.171	0.019
2	0.222	0.092	0.198	0.186	0.048
3	0.244	0.101	0.213	0.193	0.043
4	0.232	0.094	0.205	0.181	0.046
5	0.213	0.063	0.213*	0.164	0.027
6	0.225	0.065	0.225*	0.176	0.027
7	0.220	0.099	0.200	0.176	0.046
8	0.208	0.082	0.208*	0.181	0.043
9	0.232	0.089	0.208	0.186	0.043
10	0.246	0.080	0.246*	0.188	0.046
Mean	0.225	0.081	0.210	0.180	0.039
Std. Dev.	0.0133	0.0175	0.0163	0.0087	0.0105

* the original KB was returned as best.

Fault 1: Wrong binder weight
Fault 2: Wrong filler
Fault 3: Wrong target-tablet-weight

Table 8.2: Error rates for refined TFS-1A KBs

The error rates vary between columns, but do not vary greatly within each column. The reason is that when KRUST is given a refinement case exhibiting a particular fault, it will generate a refinement fixing that fault, but no others. Therefore, the error rate for the refined KB will depend on the frequency of the fault; the more frequent the fault, the

lower the error rate. Thus, because the binder fault is the most common, the remaining error rate for KBs which fix this fault is the lowest. Conversely, the filler fault is the least common, so KBs which fix only this fault have the highest remaining error rate.

The improvement for the separate fault types, and for the combined refinements, is significant at the 95% level. The mean error rate obtained by combining the three refinements in each row is small (3.9%), compared to the mean error rate for the original KB (22.5%) but non-zero. The reason that an error rate of zero is not obtained is the inability of KRUST to learn precisely the value for the maximum level of calcium phosphate, as described earlier.

Having discussed some overall properties of the results, the remainder of this section is concerned with more detailed observations about KRUST's repairs for the various fault types.

The wrong-filler fault had a second interesting property, in addition to the difficulty of learning the correct value for the missing fact. As stated earlier, KRUST always *generates* a KB which fixes the refinement case. However, this KB is not always returned as the best KB. There were four occasions when the KB containing the "correct" refinement for the wrong-filler fault had the same accuracy as the original KB, so the original KB was returned by KRUST. Two factors combined to produce the poor performance of the refined KB.

- There were very few examples of the fault (fault 2, wrong filler) in the training set, so that the improvement in overall accuracy resulting from fixing this fault was small.
- When some examples of fault 3 (wrong target-tablet-weight) were used as judging examples, the refined KB performed *worse* than the original KB. This is because an error in the target-tablet-weight seriously disrupts the rest of the formulation process, so that changes to the KB can have an unpredictable effect on the result.

This illustrates the potential importance of the choice of judging examples in the refinement process, which is discussed in detail in (Palmer & Craw 1996).

The number of refinements generated depended on the fault, but there was almost no variation between runs. Table 8.3 shows the numbers of filtered refinements and refined KBs for each fault type. In the entire run, there was a single exception to these figures; one example of fault 1 generated only 9 refinements, and 6 refined KBs. The numbers of

	Fault 1	Fault 2	Fault 3
#refinements after filtering	27	40	31
#refined KBs before filtering	6	9	6
#refined KBs after filtering	1	1	1

Table 8.3: Numbers of refinements at different stages of the refinement process

refinements after filtering are of the same order as those produced by applying the original KRUST to the wine advisor, which were: average 19.05, minimum 5, maximum 38, over the entire set of runs recorded by Craw (1991). The number of refined KBs before filtering was not recorded by Craw (1991), so no comparison can be made. However, there is a considerable difference between the wine advisor and TFS-1A in the number of refined KBs *after* filtering. For TFS-1A, there is only ever one KB which performs correctly for the refinement case. For the wine advisor, the average number is 8.42. It is likely that this difference arises from the fact that the wine advisor is a classificatory system, whereas TFS is a design system. The output of TFS is much more complex than that of the wine advisor. Consequently, it is much less likely that an incorrect TFS KB will generate the correct output for the refinement example than that an incorrect wine advisor KB will. Consequently, the refinement case filter is much more effective for TFS than for the wine advisor.

8.5.2 Feedback from evaluation by expert

When the domain expert presented me with TFS-1A, he told me that it contained three faults. He thought that one should be easy to fix, one would be hard, and that KRUST would probably not be able to fix the third! Our experiences show that KRUST was able to fix all three faults. The results of this experiment demonstrate that KRUST has succeeded in performing the role for which it was intended; to fix faults actually occurring in a KBS.

8.6 Evaluation of KRUST applied to TFS-1B

It was expected that the refinement of TFS-1B would prove to be a harder problem than TFS-1A, because the upgrade from TFS-1B to TFS-2 involved a significant change in the system's specification. Consequently, it was not surprising that, although KRUST generated a number of correct changes to TFS-1B, it was less successful overall than

when applied to TFS-1A. This section first describes how the experiment was carried out. It then presents some typical refinements generated by KRUST for various types of faults, together with comments from our expert pharmacist. He confirms whether or not the changes introduced by KRUST bring TFS-1B closer to TFS-2; where they do not, he assesses to what extent KRUST's refinement are reasonable alternatives. Statistical results are then presented in the same form as for TFS-1A. Finally, conclusions are drawn about the effectiveness of KRUST, and how its performance might be improved.

8.6.1 Experimental Design

The number of discrepancies between system and oracle output was much greater for TFS-1B than for TFS-1A. There were very few examples for which the formulations were entirely in agreement, and in many cases they differed in a number of attributes. Because of this complexity, only examples having single incorrect attributes were used in the evaluation. Errors were observed for four different excipient types; filler, binder, disintegrant and surfactant, and in the target tablet weight.

KRUST was applied to TFS-1B in a similar way as to TFS-1A. Ten iterations were performed, and in each iteration KRUST was run on 5 different refinement examples, where each example exhibited a fault in a different attribute. For each refinement example, the best refined KB was evaluated on the testing examples. The best refinements for each fault were then combined in a single KB, and this also evaluated on the testing examples.

8.6.2 The difficulties of the maintenance task

The differences between TFS-1B and TFS-2 were of two types: minor adjustments that could be regarded as bug-fixes, and more substantial changes that could better be described as maintenance. The claim that the addition of excipient categories constitutes software maintenance, and not simply debugging, is justified for the following reasons.

- The changes took several man-months to carry out.
- The expert regarded the changes as constituting a "paradigm shift" in the way in which formulation was carried out.
- The changes meant that the formulations generated by TFS-2 differed from those generated by TFS-1B for the great majority of possible inputs.

KRUST was able to make a number of minor adjustments correctly, but was less successful at generating the more substantial changes. An important reason for this lay in the nature of the formulation problem. In general, there is no one best formulation, but rather a number of equally good ones. This poses problems for judging and evaluation, since the only criterion for evaluating a formulation is by comparing it with the oracle's. Consequently KRUST has no way of distinguishing a refinement which leads to a good alternative formulation from a refinement which is entirely wrong. This makes it hard to select the best refined KB. As a result, KRUST will often generate a refinement that the expert agrees to be reasonable, but KRUST rejects it on the grounds that it does not perform in the same way as the oracle on the refinement example. As a result, KRUST is often unable to recommend a refined KB, so returns the original KB as best. Evaluation of KRUST's performance is difficult for the same reason; a simple comparison with the oracle does not tell the whole story.

It will be noted that the refinement of TFS-1A was affected less by this problem, since KRUST was able to refine TFS-1A in such a way that its behaviour corresponded very closely to that of TFS-1B. This may be explained by the fact that the refinements required by TFS-1A were simpler. In particular, less use was made of the inductive operators, since neither **inductive_adjust_value** not **inductive_split_rule** were needed in the refinement of TFS-1A. The approach taken by the inductive operators renders them particularly liable to the problem of generating alternative good solutions, since they work by identifying features which distinguish correct from incorrect behaviour over a number of examples; they therefore identify a whole range of good behaviour, where good behaviour may correspond to the choice of any one of a number of suitable excipients.

There follows a description and evaluation of the refinements actually generated by KRUST, classified by fault type. For faults where KRUST was unable to generate refinements which duplicate the behaviour of TFS-2, the description concentrates on the refinements generated by the inductive operators, particularly **inductive_split_rule**.

8.6.3 Target-tablet-weight

In TFS-1B, as in TFS-1A, the target-tablet-weight is a function of dose only. Two different formulae are used, depending on whether the dose is less than or greater than 350 mg. The following two rules perform the calculation.

```
RULE 1ST-GUESS-WEIGHT-<=350MG
IF    DRUG has value <DRUG> in the FORMULATION
 AND  <DRUG> has value <DOSE> in the FORMULATION
 AND  <DOSE> ≤ 350
 AND  <WEIGHT> = ROUND-TO-NEAREST-5(100 * <DOSE>
                 /(0.221 * <DOSE> + 10))
THEN
 set the value of TARGET-TABLET-WEIGHT in
                 SPECIFICATION to be <WEIGHT>

RULE 1ST-GUESS-WEIGHT->350MG
IF    DRUG has value <DRUG> in the FORMULATION
 AND  <DRUG> has value <DOSE> in the FORMULATION
 AND  <DOSE> > 350
 AND  <WEIGHT> = ROUND-TO-NEAREST-5(3.1317 + 0.375 * <DOSE>
                 - 0.000544 * <DOSE>^2 + 2.53 * 10^-7 * <DOSE>^3)
THEN
 set the value of TARGET-TABLET-WEIGHT in
                 SPECIFICATION to be <WEIGHT>
```

For examples exhibiting a target-tablet-weight fault, the recommended refinement consists of two applications of the **inductive_adjust_value** operator. This operator is used to specialise the threshold condition to <DOSE> $\leq$ 350 in the first rule, and to generalise the equivalent condition in the second rule. The effect of these changes is to lower the threshold at which the second rule is applied from 350 mg to 310 mg.

The expert confirmed that the changes made in the rules which calculate target-tablet-weight were correct. The refined rules generated by KRUST are the same as those used by TFS-2.

8.6.4 Surfactant

For all the examples of incorrect surfactant, the nature of the fault was that TFS-1B recommended some surfactant, but the oracle indicated that no acceptable surfactant

could be found. The refinement selected by KRUST for all such examples was to delete the rule DEFAULT-SURFACTANT.

```
RULE DEFAULT-SURFACTANT
IF    FORMULATION has attribute <SURFACTANT>
THEN
 set the value of SURFACTANT in
      FORMULATION to be <SODIUM-LAURYL-SULPHATE>
UNLESS
      SURFACTANT has value <VALUE> in FORMULATION.
```

The effect of this rule is to choose sodium lauryl sulphate as a surfactant, provided that a surfactant is required, and that none has yet been chosen. Deleting this rule causes TFS-1B's behaviour in choosing surfactants to match that of the oracle, so that in certain circumstances it fails to recommend any surfactant.

The expert confirmed that this refinement was correct; the rule DEFAULT-SURFACTANT is not present in TFS-2.

8.6.5 Binder

For the binder and other fault types, the refinements generated by KRUST were less successful, and were mostly unable to duplicate the exact behaviour of the oracle. Unless stated otherwise, it may be assumed that all the refinements listed below were rejected because they did not perform identically with the oracle for the refinement case. For binder faults, the refinements generated by KRUST are listed in some detail, to illustrate the behaviour of the inductive operators. The refinements for later faults are presented more briefly.

For the examples where the fault lay in the binder, KRUST generated a number of refinements using the **inductive_split_rule** operator, and on some occasion the **inductive_adjust_value** operator as well. On one occasion, the first of these operators generated a KB which gave the same result as the oracle on the refinement case. The **inductive_split_rule** operator was applied to the rule MOST-STABLE-BINDER, and created

four copies of the rule, each of which had extra conditions. MOST-STABLE-BINDER sets the binder to be the one which appears at the top of the stability agenda.

```
RULE MOST-STABLE-BINDER
IF    <BINDER> is on <STABILITY-AGENDA>
      <BINDER> is-a BINDER
THEN
      set the value of BINDER in
                FORMULATION to be <BINDER>
UNLESS          BINDER has value <VALUE> in FORMULATION.
```

---

inductive_split_rule creates four new rules by adding the following sets of conditions to the MOST-STABLE-BINDER rule.

```
      STABILITY of MANNITOL < 100.9
      STABILITY of MAIZE-STARCH < 101.1
      STABILITY of CALCIUM-PHOSPHATE < 39.54
      STABILITY of CALCIUM-DIHYDRO-PHOSPHATE < 92.15
      NO-OF-FILLERS of SPECIFICATION < 1.5
      DOSE of SPECIFICATION ≥ 235
      CATEGORY of <BINDER> < 2.5

      STABILITY of MANNITOL < 100.9
      STABILITY of MAIZE-STARCH < 101.1
      STABILITY of CALCIUM-PHOSPHATE < 39.54
      STABILITY of CALCIUM-DIHYDRO-PHOSPHATE ≥ 92.15
      DOSE of SPECIFICATION ≥ 235
      CATEGORY of <BINDER> < 2.5

      STABILITY of MANNITOL < 100.9
      STABILITY of MAIZE-STARCH < 101.1
      STABILITY of CALCIUM-PHOSPHATE ≥ 39.54
      CATEGORY of <BINDER> < 2.5

      STABILITY of MANNITOL ≥ 100.9
```

The behaviour of these new rules is as follows. The first new rule will attempt to fire for each in turn of the binders on the stability agenda. The rule will succeed if all the conditions are satisfied, and will select the successful binder to be included in the formulation. If the first rule does not succeed for any binder, the second rule will be tried, and so on. The exception clauses mean that as soon as a binder has been chosen, none of these binder-selecting rules can fire further. The effect of the rules is thus to select binders which satisfy one of the above sets of conditions in preference to those that do not.

Note that a condition such as STABILITY of MANNITOL < 100.9 refers to the stability of mannitol *with respect to the current drug*, so is a drug property. Such a condition might initially appear meaningless in a rule designed to choose the best binder. However, on further inspection it turns out that the induced rules do make sense. Their effect is to prefer binders with category < 2.5 in certain circumstances only. These circumstances are that the current drug has certain chemical properties, which are revealed by the relative stability of certain excipients with respect to that drug. Nonetheless, it must be admitted that a human expert would not normally use as a drug property the stabilities of excipients *not* included in the formulation.

For other binder faults, KRUST generated a successful refinement which was much simpler. This was to elevate the priority of the rule DEFAULT-BINDER over that of MOST-STABLE-BINDER.

```
RULE DEFAULT-BINDER
IF    FORMULATION has attribute <BINDER>
THEN
      set the value of BINDER in
            FORMULATION to be <PREGELATINISED-STARCH>
UNLESS
      BINDER has value <VALUE> in the FORMULATION.

RULE MOST-STABLE-BINDER
IF    <BINDER> is on <STABILITY-AGENDA>
      <BINDER> is-a BINDER
THEN
```

```
    set the value of BINDER in
            FORMULATION to be <BINDER>
UNLESS
    BINDER has value <VALUE> in the FORMULATION.
```

The effect of MOST-STABLE-BINDER is to choose the binder with the highest stability. The effect of DEFAULT-BINDER is to choose pregelatinised-starch if no other binder has been recommended. This effect of promoting the default rule is therefore to choose pregelatinised starch much more frequently. This fixes a number of examples in which the oracle chooses pregelatinised starch, but TFS-1B does not.

Finally, KRUST was also able to fix a number of faults by altering the conclusion of the rule SOLUBLE-INGREDIENTS-RULE. This rule asserts that in certain conditions, the binder should be maize starch. KRUST changes the conclusion to choose the binder pvp.

The expert's assessment

The expert confirmed that KRUST was correct to learn the condition that category of binder < 2.5, and that this corresponded to the new policy of preferring binders with category 1 or 2. However, he said that the additional conditions based on the stabilities of unrelated excipients were not sensible, especially when excipients were included which were not even binders. However, given the presence of large numbers of different excipients in the database, it was not surprising that patterns in the data would sometimes emerge which would cause such conditions to be learned. He suggested that expert knowledge was needed to identify in advance which attributes were potentially relevant and which were not. The need for expert knowledge arose again when discussing filler faults.

The expert confirmed that changing the right-hand side of SOLUBLE-INGREDIENTS-RULE from maize starch to pvp is reasonable, and that it brought the behaviour of TFS-1B closer to that of TFS-2. However, it would have been better to learn a more general rule which made use of excipient categories.

8.6.6 Disintegrant

KRUST used the **inductive_split_rule** operator to generate a number of refinements for disintegrant faults. The operator generates fewer new rules, and adds fewer conditions, than when applied to binder faults. A typical refinement is to the rule GET-BEST-DISINTEGRANTS.

```
RULE GET-BEST-DISINTEGRANTS
IF    <DISINTEGRANT> is on DISINTEGRANT-AGENDA
      STABILITY of <DISINTEGRANT> ≥ 90
THEN
 set the value of DISINTEGRANT in
                FORMULATION to be <DISINTEGRANT>
```

The effect of this rule is to choose the first disintegrant on the DISINTEGRANT-AGENDA that has stability greater than 90. The **inductive_split_rule** operator creates a higher-priority copy of this rule with the added condition

EFFICIENCY-RATING of <DISINTEGRANT> > 9

The effect of this new rule is to select disintegrants with high efficiency in preference to the rest, provided that they also have sufficiently high stability. On another occasion, the **inductive_split_rule** added a different condition:

INITIAL-LEVEL of <DISINTEGRANT> < 0.045

For a different refinement example, also suffering from a wrong binder, KRUST generated an alternative refinement to the rule GET-BEST-DISINTEGRANTS. In this case, the **inductive_adjust_value** operator increased the stability threshold from 90 to 102, so that the second rule condition appeared as

STABILITY of <DISINTEGRANT> ≥ 102

Finally, on several occasions, KRUST deleted the rule DEFAULT-DISINTEGRANT, but this did not cause TFS's behaviour to match that of the oracle.

The expert's assessment

The expert confirmed that adding a condition to GET-BEST-DISINTEGRANT to prefer disintegrants with high efficiency was a reasonable change to make. Adding the condition that the initial-level of the disintegrant should be low was also understandable, since the higher the efficiency of a disintegrant, the lower its initial-level. Therefore, selecting disintegrants with low initial-level is an indirect way of obtaining efficient ones. However, a human formulator would be more likely to use efficiency directly as the criterion.

The increase in the stability threshold was not approved by the expert. He agreed that high stability excipients were generally preferable, but said that there was nothing to be gained by raising the threshold above the present value of 90. However, he agreed that the resulting change in the disintegrant selected by TFS was an improvement.

Finally, the expert confirmed that the refinement which deleted the DEFAULT-DISINTEGRANT rule was correct. However, the presence of other faults in TFS-1B meant that the rule deletion on its own was not sufficient to produce correct behaviour.

8.6.7 Filler

The nature of the refinements generated by KRUST for filler faults depended on whether the formulation was for one or two fillers. For one filler formulations, KRUST applied **inductive_split_rule** on a number of occasions. It also recommended deletion of the rule INSOLUBLE-FILLER-RULE as an alternative refinement. This rule asserts that if the drug is soluble, the filler must be insoluble.

For two-filler formulations, the **inductive_split_rule** operator generated a number of refinements to the rule CHOOSE-A-FILLER-PAIR. As for the binder faults, the conditions added include a number of references to the stability of various excipients with respect to the drug being formulated.

The expert's assessment

The expert initially thought that KRUST was correct to delete INSOLUBLE-FILLER-RULE, and he confirmed that this change resulted in a better choice of filler in a number of cases. However, it turned out that INSOLUBLE-FILLER-RULE does still exist in TFS-2, but that for the examples in question it is overridden by other TFS-2 rules. These new rules

insist that certain physical properties of the tablet, viz. tablet-yp and tablet-srs, should not exceed certain thresholds. When the requirements of these tablet-property rules are incompatible with INSOLUBLE-FILLER-RULE, the tablet-property rules takes precedence. KRUST is currently unable to learn such rules, since it does not use intermediate results such as tablet-yp as input to the induction algorithm. Consequently, deleting INSOLUBLE-FILLER-RULE was the nearest it could come to the correct refinement. The expert pointed out that KRUST would have benefited from knowing that tablet properties are an important intermediate result which is used by TFS-2.

The new tablet-property rules made KRUST's task harder in another way. When the expert first looked at the TFS-2 formulations, he was very surprised at the number of category 3 fillers which were included. It turned out that these were explained by the tablet-property rules, which take precedence over the policy of preferring low category excipients. These category 3 fillers in TFS-2's formulations made it harder for KRUST to induce rules which prefer low category fillers, since category did not appear to be a good attribute for predicting which fillers would be recommended. This explanation is confirmed by the observation that during earlier experiments using small numbers of examples, KRUST *was* able to learn the condition that the category of filler should be ≤ 2. This was possible because the example set happened not to include any of the misleading examples for which TFS-2 recommended a category 3 filler. This observation suggests that the performance of the inductive operators might be improved if expert information were provided about which examples were typical and which were outliers.

The expert made a further point about TFS-2's rules for selecting fillers. The rules for the two filler case are particularly complex, because the relative ranking of pairs of fillers of different categories have to be calculated. For example, is it better to choose a category 1 filler and a category 3 filler, or two category 2 fillers? This further explains KRUST's poor performance in this area.

The principal conclusions from the results for filler faults are that KRUST needs an expert to suggest which attributes are appropriate for inclusion in new rule conditions, and that the presence of one serious fault in TFS can make it harder to fix other faults.

8.6.8 Statistical results

Table 8.4 summarises the results for the application of KRUST to TFS-1B. Starred entries indicate that the original KB was returned as best.

	Error rates for KB						
	Original KB	Best KB, when refinement case is taken from the given fault class					KB combining refinements
Run		Fault 1	Fault 2	Fault 3	Fault 4	Fault 5	
1	0.341	0.300	0.307	0.341*	0.341*	0.341*	0.275
2	0.287	0.263	0.273	0.287*	0.287*	0.287*	0.251
3	0.312	0.278	0.300	0.285	0.312*	0.312*	0.258
4	0.309	0.295	0.295	0.283	0.309*	0.309*	0.278
5	0.331	0.304	0.316	0.331*	0.331*	0.331*	0.297
6	0.326	0.326*	0.307	0.307	0.326*	0.326*	0.295
7	0.321	0.304	0.307	0.321*	0.321*	0.321*	0.297
8	0.316	0.316*	0.297	0.316*	0.316*	0.316*	0.297
9	0.353	0.324	0.338	0.353*	0.353*	0.353*	0.319
10	0.331	0.302	0.316	0.290	0.331*	0.331*	0.278
Mean	0.323	0.301	0.306	0.311	0.323	0.323	0.285
Std. Dev.	0.0183	0.0195	0.0168	0.0251	0.0183	0.0183	0.0203

* the original KB was returned as best.

Fault 1: Wrong target-tablet-weight
Fault 2: Wrong surfactant
Fault 3: Wrong binder
Fault 4: Wrong disintegrant
Fault 5: Wrong filler

Table 8.4: Error rates for refined TFS-1B KBs

This table summarises the results presented above. As with TFS-1A, given a refinement example exhibiting a particular fault, KRUST's refinements fix faults of that type only. Moreover, in the case of TFS-1B there may be a number of faults each of which affect the choice the same excipient type, but normally only one such fault is fixed at a time. Since TFS-1B suffers from so many faults, the potential improvement in accuracy resulting from each individual refinement is smaller than for TFS-1A, as the results show.

For most examples of the target-tablet-weight, and for all examples of the surfactant faults, KRUST was able to generate refinements which fixed the refinement example and all other examples of the same fault. For some examples of binder faults, KRUST was able to generate refinements which fixed the refinement example and some other examples of

the same fault. For the other fault types, although KRUST was able to generate a number of refined KBs which were acceptable to the expert, it had no way of realising this, and so rejected them and returned the original KB as best. For faults 1, 2 and 3, and for the KB combining the refinements, the improvements resulting from KRUST's application are significant at the 95% level.

8.6.9 The expert's assessment

After examining all these results, the expert concluded that KRUST had identified the majority of the differences in behaviour between TFS-1B and TFS-2, and had suggested reasonable rule changes for many of them. In some cases, KRUST had "taken a step in the right direction, but had not gone all the way".

The expert emphasised the magnitude of the change that had been introduced in TFS-2. It was a significant policy shift, and not just the refinement of pharmacological knowledge. In fact, the new rules were based not on pharmacology, but on industrial requirements external to the formulation process, such as the need to maintain consistency within an international organisation, and to reduce inventory costs. Hence the formulations generated by TFS-2 might actually be poorer when considered in isolation than those generated by TFS-1B. The expert believed that the nature of the shift would make it particularly difficult to duplicate automatically, but that KRUST had made a good attempt.

8.6.10 Conclusions from the expert's remarks

The principal conclusion from the expert's assessment of KRUST's performance is that KRUST would be more effective if it were given expert advice in two areas.

- Identifying which attributes are likely to be useful when learning new rule conditions.
- Selecting good refinements.

Assistance is required in selecting refinements for two reasons.

- A refinement may cause an improvement in the behaviour of TFS, but because it still does not match the oracle, KRUST has no way of determining that the refinement is a good one, so needs an expert's assistance with refinement selection.

- In complex situations where TFS has undergone many rule changes, KRUST may be able to duplicate these changes only one at a time. In this situation, the first change made by KRUST may result in no apparent improvement at all. Here again, an expert is required to confirm that the refinement is a good one.

Chapter 9

Comparisons with other refinement systems

This chapter compares KRUST with two major competitors, EITHER (Ourston & Mooney 1994) and CLIPS-R (Murphy & Pazzani 1994). EITHER was chosen as a sophisticated representative of other refinement systems. The comparison with EITHER concentrates on the use of inductive operators, since this is the principal area of the work described in this thesis for which a corresponding feature exists in EITHER. CLIPS-R was chosen because it is the only system apart from KRUST to refine forward-chaining rules. Moreover, the authors share my aim of tackling the problems of refining industrial expert systems.

9.1 A comparison of inductive operators in KRUST and EITHER

A significant contribution made by this thesis is the implementation of inductive operators for KRUST. To some extent, this was done in order to correct a weakness, and provide KRUST with an ability which other tools already possessed. However, there are some differences between KRUST's inductive operators and those of other systems. In this section, KRUST's inductive operators are compared with those of EITHER.

9.1.1 The inductive_adjust_value operator

EITHER possesses a similar operator to KRUST's **inductive_adjust_value**, which will generalise the threshold in a numerical comparison just far enough to cover failing negatives; that is, for examples for which the rule should have fired and did not. Similarly, it will specialise the threshold just far enough to cover failing positives. The approach taken by KRUST is similar, but more general. First of all, EITHER refines classificatory systems, whose output for any given example is simply a class value. KRUST refines systems which generate a complex output for each example. Consequently the selection of examples from which to learn is a harder task for KRUST than for EITHER. Positive examples for generalisation can not be characterised simply as false negatives, since there is no simple concept of negative and positive. Instead, the positive example used by KRUST when generalising a condition are characterised as "all examples which suffer from the same fault and for which the condition being refined failed". Similarly, positive examples for specialisation are characterised as "all examples which suffer from the same fault and for which the rule being refined succeeded".

9.1.2 The inductive_split_rule operator

This operator may be compared with EITHER's inductive rule addition operator. There are a number of differences between the two operators. First of all, their purposes differ. **inductive_split_rule** aims to alter the order in which objects are chosen, so that a different object is chosen first. To do this, it makes modified copies of a rule with higher priority than the original rule. This is an appropriate modification to a design system, where it is often the case that one choice is better than another, rather than one choice being entirely right or entirely wrong. It also illustrates KRUST's ability to reason about and manipulate the actual behaviour of rules, including the order in which they fire. The purpose of EITHER's rule addition operator is different. It seeks to add a rule so that a conclusion will be true for a particular example which was not true before. One consequence of this difference is that, since KRUST seeks to change the order of rule firing only, it retains all the conditions of the original rule and adds further conditions to the copies. This means that the new rule set, consisting of the original rule plus some modified copies, will fire in exactly the same circumstances as the original rule on its own. On the other hand,

EITHER seeks to generate a rule which will fire when the old rule did not. Consequently, it creates a new rule which does not retain the conditions of the old rule.

A second difference is that, whereas EITHER's operator seeks to enable a conclusion which was previously unprovable, KRUST's operator seeks to enable a conclusion which *was* previously provable, but with different bindings. For example, EITHER's operator might seek to make the query `insoluble(calcium phosphate)` succeed, where before it failed. On the other hand, KRUST's operator might seek to make the query `filler of formulation is <filler>` succeed, binding <filler> to calcium carbonate, whereas before it succeeded with filler bound to calcium phosphate.

This difference has a bearing on the properties from which new conditions are induced. EITHER uses example properties, both observables and intermediate results. KRUST currently uses observables, which in the case of TFS are drug, dose and number of fillers, together with features which may be regarded as one step removed from observables, such as the properties of the drug. However, it also uses properties of the object which the rule is recommending; that is, the object to which a variable in the rule's conclusion becomes bound when the rule fires. In the case of TFS, this object is normally an excipient. However, the KBs to which EITHER has been applied do not appear to include such "object-generating" rules, so this particular source of information is not available to it.

To summarise, KRUST's **inductive_split_rule** and EITHER's inductive rule addition operator have slightly different purposes and use different sources of information. KRUST's operator appears to be performing a more difficult task, since KRUST's operator is concerned with the variable bindings for which a rule fires, and with the orders in which rules fire, and EITHER's operator is concerned with neither of these things.

9.1.3 The inductive_add_fact operator

This operator may in a sense be regarded as a special case of EITHER's rule addition operator, since it learns a new rule with no conditions. However, it differs from EITHER's operator in that the purpose of KRUST's new rule is again not just to satisfy a previously unsatisfied condition elsewhere, but to satisfy it for a particular set of variable bindings. For example, if the condition `max-level of <filler> is <max>` fails, the operator will add a fact which will cause the condition to succeed *for particular values* of `<filler>` and `<max>`.

9.1.4 The use of intermediate results

As mentioned above, EITHER is able to use any intermediate result when inducing new rules, and KRUST uses only a limited class of intermediate results. The extra search this imposes on EITHER is balanced by the fact, that unlike KRUST, it uses induction only when all other operators have failed.

The expert's analysis of KRUST's performance presented in section 8.6.9 indicated that KRUST might perform better if it did use certain intermediate results. He felt that specialised domain knowledge was needed in selecting the attributes, either observables or intermediate, to be used as input to the induction algorithm. In the case of TFS, he believed that KRUST should not have used the relative stabilities of excipients with the drug being formulated, and that it should instead have used certain physical properties of the tablet.

9.2 A comparison of KRUST and CLIPS-R

This section describes CLIPS-R (Murphy & Pazzani 1994). CLIPS-R refines KBSs written in CLIPS (Girratano & Riley 1994), and is the only system apart from KRUST to refine forward-chaining rules. Moreover, the authors share my aim of tackling the problems of refining industrial expert systems. Consequently, it is instructive to compare the approaches of CLIPS-R and KRUST to similar problems. This chapter describes the sequence of operations performed by CLIPS-R, and where appropriate, draws comparisons with KRUST. Once the description of CLIPS is complete, further comparisons with KRUST are made.

The features of CLIPS KBSs which Murphy & Pazzani (1994) regard as distinctive about CLIPS, and particularly significant for the refinement task are:

- forward-chaining rules;
- the ability of rules to retract facts;
- the ability of rule firings to cause interactions with the user as side-effects.

The chaining direction of rules mainly affects CLIPS-R's blame allocation. The ability of rules to retract facts affects refinement generation, but can be handled by a straightforward extension of an algorithm which handles assertions. The side effects of rule firings

affect both blame allocation and refinement generation. PFES KBSs share the first two features: forward-chaining rules and (effectively) the ability to retract facts, so these must be handled by KRUST as well as CLIPS-R. However, KRUST has not so far been concerned with the side effects of rules in a KBS, mainly because the KBSs to which it has been applied have not had a major interactive element.

9.2.1 Blame allocation in CLIPS-R

CLIPS-R uses a much wider definition of the *behaviour* of a KBS than other refinement systems. The user can specify two aspects of a KBS behaviour:

- the content of working memory (the fact-list) when execution halts, and
- the order in which observable actions are carried out.

The constraints that the user can place on the final fact-list take the form of requirements that a certain fact must or must not be present in memory. An observable action, also known as a side-effect, is an action which displays information, or requests information from the user. The specification for the order of observable actions takes the form of a finite state machine. The ordering constraints for an example are said to be satisfied if the sequence of observable actions generated by the example is accepted by the finite state machine associated with that example. If the sequence is *not* accepted by the machine, then the number of violations in the sequence is defined to be the minimum number of additions and deletions required to make the sequence acceptable.

When an example is run, it is assigned an *error rate*, which is defined to be the number of constraint violations divided by the total number of constraints.

At the start of the blame allocation phase, CLIPS-R runs a CLIPS KBS on the available training examples, and arranges the resulting traces into a trie-structure. Figure 9.1 shows an example of the trie-structure for the student loan advisor (Pazzani 1993). This is a KB which determines whether a student should repay a US educational loan. The only trace information used at this stage is a list of which rules fired, and in which order. The trie-structure is "a compact representation of rule traces that groups together those instances that share an initial sequence of rule firings" (Murphy & Pazzani 1994). Each line in figure 9.1 represents a node of the trie structure, which in turn is associated with a particular initial sequence of rule firings. Each node is then associated with the set of

```
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 9.1: A trie-structure created by CLIPS-R for the student loan domain

examples which gave rise to that sequence of rule firings. The three numbers at each node represent respectively the total number of errors (constraint violations) for all the examples at that node, the total number of constraints, and the ratio of the two, which represents an error rate. Each leaf in the trie structure represents a point at which rule-firing ceased for one or more traces.

The purpose of building the trie-structure is to determine the class of examples sharing a common initial rule firing sequence which suffer from the greatest proportion of errors. When CLIPS-R has built the trie structure, it identifies the leaf node with the highest error rate, and then tries to fix the examples associated with that node. For each constraint violation of each example at this node, a set of refinements is generated (section 9.2.2).

Comparison with Krust

The only aspect of a system's behaviour with which refinement systems are usually concerned is the class to which a KBS assigns an example. KRUST extends this idea to cover the complex output generated for an example by a design system. CLIPS-R appears to extend the idea still further when it considers the entire content of final working memory. However, those CLIPS facts which appear in working memory, but are not part of the system output, are either initial facts or intermediate results. Palmer & Craw (1995) have shown that KRUST is able to generate refinements based on intermediate results. Hence,

in this respect, the two refinement systems are equally flexible. (The work on the use of intermediate results was not carried out by me).

On the other hand, the ability of CLIPS-R to handle constraints on a KBS's order of actions is not shared by KRUST. Experience to date, both of myself and others, with applying KRUST to a number of KBSs has not indicated any requirement to impose constraints on the order of actions performed by a KBS. However, another reason for this lack may be that when KRUST is applied to a KBS, the first step has been replace all user interactions with database accesses or some other kind of programmer interface. This means that errors in user interactions are less likely to be used to drive refinement.

9.2.2 Refinement generation

Once constraint violations have been identified for a number of examples, as specified above, CLIPS-R generates refinements with the aim of correcting them.

CLIPS-R is provided with a table which lists appropriate refinement types for the various possible faults exhibited by an example. The faults exhibited by an example will be violations of the constraints on the final fact list, so will either be of type *Extra Fact* or *Missing Fact*[1]. However, the refinement generation procedure may subsequently suggest other fault types which may have caused the exhibited faults. For example, a *Missing Fact* fault may have been caused by a *Missing Rule Firing*. Here are the refinements for the fault *Missing Rule Firing*.

Fault: Missing Rule Firing

- Generalise LHS of rule.
- Increase salience (priority) of rule.
- For each unsatisfied unnegated conditional element in the rule, identify repairs that could allow a fact which will satisfy the condition to be present in the fact-list at the time that the rule should have fired.
- For each unsatisfied negated conditional element in the rule, identify repairs that could cause all facts that prevented the conditional element from being satisfied to be missing from the fact list.

[1] Violations of ordering constraints are considered later

These refinements correspond to KRUST's refinements for a No-Fire and Can-Fire rules. The possible classes of refinement just listed may be summarised in more KRUST-like terms as: to increase the rule's priority, to refine one or more conditions, or to modify the behaviour of *other* rules in order to enable one or more of this rule's conditions. Thus CLIPS-R works *backwards* from a faulty conclusion, identifying possible faults in the rule chain leading to that conclusion, in the same way that KRUST does.

The handling of negation

CLIPS includes two forms of negation, which complicate the refinement process: explicit negation in rule conditions, and retract operators in rule conclusions. Negated rule conditions are covered by the final paragraph in the list of refinements quoted above for *Missing Rule Firings*, which explains how CLIPS-R fixes unsatisfied negated conditions.

The description of repairs in this list is also general enough to cover both fact assertions and fact retractions. A repair "that could allow a fact to be present" may be achieved either by enabling a rule which asserts the fact, or by disabling a rule that retracts the fact. The repairs for negated conditions can be achieved similarly, *mutatis mutandis*.

PFES's `remove-from-agenda` plays a similar role in rule chaining as CLIPS's `retract` operator, so it is not surprising that KRUST handles `remove-from-agenda` in the same way that CLIPS-R handles `retract` (section 5.5.1).

Correcting ordering constraint violations

If a KBS executes observable actions for some example in an order that can not be generated by the finite-state machine associated with that example, then the constraints have been violated. These violations are repaired by identifying actions which should be added to or deleted from the sequence. The violations can thus be expressed as *Missing Fact* or *Extra Fact* faults, and then fixed as described above.

9.2.3 Use of induction

The actual refinement operators which carry out the repairs do not differ significantly from the operators used by other RSs, so the only operators described here are those that use induction. CLIPS-R makes use of two inductive techniques. First, a specialisation

operator is able to learn a new condition to add to an over-general rule. Second, induction is used at the refinement implementation stage to learn a new rule, if no other operator gives a decrease in the error rate.

The use of induction to specialise a rule

CLIPS-R is able to specialise the LHS of a rule by adding a new rule condition. To obtain a set of new conditions from which to choose, CLIPS-R first lists all the conditions existing in the initial KB and examples. It then creates an exhaustive set of condition templates by replacing constants with variables in these conditions in all possible ways. Finally, it tests these conditions by adding each in turn to the over-general rule and evaluating the resulting KB.

The use of induction to learn a new rule

When all the other refinement operators fail, the rule induction operator is invoked. A number of rules are generated, then revised through a hill-climbing iterative refinement search. The LHS of each new rule is generated by selecting pairs of examples with the same constraint violations, and taking the least-general generalisation (LGG) (Plotkin 1971) of their initial fact lists. If the common constraint violation is a missing fact, the RHS of the new rule asserts that fact; if the violation is a superfluous fact, then the RHS of the new rule retracts that fact.

9.2.4 Further comparisons of Krust and Clips-R

A further difference between CLIPS-R and KRUST is the amount of tracing information required. CLIPS-R uses all the information KRUST does, together with two other items: the fact-list prior to each rule firing, and information linking each fact to the rule that asserted it.

The presence in CLIPS traces of information about which rule asserted any given fact does not constitute a significant difference between CLIPS and PFES traces, since the same information can be derived from the PFES trace, though it is not included explicitly. Similarly, the inclusion of the fact list before each rule firing makes the refinement process easier for CLIPS-R than for KRUST, but again the missing information can be derived

from elsewhere in the trace. An example from section 5.5.1 illustrates this. Rule REMOVE-EXCESSIVE-FILLERS fails to fire for calcium phosphate, because its second condition fails. KRUST can correct this by adding a value for the maximum level of calcium phosphate to the database. However, in order to determine bounds on the maximum level, KRUST needs to know the value of FILLER-CONCENTRATION in the third condition. This value can not be determined from the trace for REMOVE-EXCESSIVE-FILLERS, since the interpreter never reached the third condition. However, the value can be determined from the trace of a rule which set the FILLER-CONCENTRATION. On the other hand, if KRUST had access to the fact-list before REMOVE-EXCESSIVE-FILLERS fired, as CLIPS-R does, it could instead have read the value of FILLER-CONCENTRATION from that list, instead of having to search elsewhere in the trace, but the result would be the same.

```
Rule REMOVE-EXCESSIVE-FILLERS
If
     <FILLER> is on FILLER-AGENDA
     MAX-LEVEL has-value <LEVEL> in <FILLER> Failed
     FILLER-CONCENTRATION has-value <CONC> in SPECIFICATION
     <CONC> is-greater-than <LEVEL>
Then
     remove <FILLER> from FILLER-AGENDA
```

9.2.5 The use of traces by CLIPS-R and KRUST

This section compares the use of traces by KRUST with three CLIPS-R procedures: the use of the trie structure, and CLIPS-R's two applications of induction in refinement generation. The trie structure is considered first.

Clips-R use of the trie structure

The effect of rule priority and conflict resolution strategies is much greater in forward-chaining than in backward-chaining KBs. Consequently, it is reasonable that the starting point for CLIPS-R's operation should be a structure, the trie structure, which is based on the order in which rules fire.

The purpose of the trie-structure to decide which faults to fix first, based on the error-

rates at its nodes. This may be regarded as an early form of refinement selection. It is clearly more efficient to select particular examples to refine rather than to generate refinements for a wider class of examples and then to reject most of them.

The creation of the trie-structure is a mechanism whereby CLIPS-R uses trace information to cluster examples. However, the mechanism and purposes for example clustering in CLIPS-R and KRUST differ in several ways. CLIPS-R

- clusters examples which have an identical sequence of initial rule firings, ignoring variable bindings, and
- uses the grouping as a mechanism for example selection.

In contrast, KRUST

- clusters examples for which one particular rule fired, with specified variable bindings. The selection also takes into account properties of the system output.
- uses trace properties of selected examples, specifically the values of variable bindings, to create new rule conditions, or to modify old ones.

To assess the broader applicability of CLIPS-R's use of the trie structure, it is worth considering if it might be applied to PFES applications. There are two potential difficulties. First, variable bindings are ignored when constructing the trie; however, it might be possible to extend the algorithm to take variables into account. Secondly, it is not clear how well the approach would fit the "generate and filter" paradigm frequently used in TFS, under which rules fire repeatedly with different variable bindings. In order to understand how a generate and filter procedure works, it is necessary to understand the chaining behaviour between the rule that generates data and the rule that filters it. However, the trie structure is not well-designed to do this, since it considers only rule firing *sequences*, not rule *chains*; while any rule in the sequence must chain with an earlier rule or fact, it need not chain with the immediately preceding rule.

It appears therefore that, although it is important to understand the order in which rules fire when refining a forward-chaining system, the approach based on building a trie-structure ignores other useful information, and so would not in its current form be applicable to a design system like TFS.

Clips-R's inductive operators

CLIPS-R's specialisation operator creates new rule conditions based on patterns observed in the initial KB, so is much less focused than KRUST's inductive operators. It is more instructive to compare CLIPS-R's rule induction operator with KRUST's.

Interestingly, when CLIPS-R groups examples for purposes of rule induction, it ignores trace information, and simply clusters examples suffering from similar faults. Examples are said to suffer from a similar fault if they violate the same constraints on the final fact-list. No account is taken of the *way* in which the constraint is violated, which would make possible a more precise characterisation of the differences between examples, as discussed in section 7.4.

A second difference is that CLIPS's rule induction operator creates an entire new rule, whereas KRUST's inductive operators just add or modify conditions. The **inductive_split_rule** comes closest to creating a new rule, since it takes copies of existing rules and then modifies the copies by adding sets of new conditions.

A last point of comparison is the manner in which CLIPS-R and KRUST create new rule conditions. The principal difference is that CLIPS-R constructs the LHS of its rule using only initial facts, which are the equivalent of observables in backward chaining, whereas KRUST also uses some intermediate results. For example, the **inductive_split_rule** induces new conditions both from initial facts, and also from properties of excipients appearing in traces which exhibit a rule's faulty behaviour. In addition, KRUST's other inductive operators make use of other variable bindings found in traces, which again correspond to intermediate results in the KBS's reasoning.

9.2.6 Conclusions

The principal difference between CLIPS-R and KRUST when applied to forward-chaining systems is that CLIPS-R takes greater account of the *order* in which rules fired. This is shown in its use of a trie-structure based on the rule-firing sequences for the example set, and by the fact that it allows the user to specify an order of actions for the KBS. On the other hand, KRUST takes account of variable bindings when clustering examples according to trace information, and so has less need to consider the order of firing. KRUST's approach therefore seems better adapted to systems which have a significant first-order element.

Chapter 10

Conclusions

This chapter recalls the goals of the project described in this thesis, and examines the extent to which they have been achieved. It then draws attention to other conclusions which may be drawn from the experience of applying KRUST to TFS. The final section proposes some directions for future work.

10.1 Principal goals

The main goals of this project were:

1. to develop a practical refinement tool, which could be applied to an industrial expert system, and which could assist with software development by identifying and suggesting fixes for faults;
2. to demonstrate that the tool was able to refine a design system, as well as classificatory ones.

Chapter 3 established that TFS is an expert system which performs a design task. It has been developed over a number of years, is in regular use in industry, and solves a hard problem. Moreover the architecture of PFES is particularly appropriate for building design tools, and differs from shells intended for classificatory tasks. Consequently, the successful application of KRUST to a PFES application would satisfy the principal goals.

Chapter 5 then showed that KRUST could be applied to TFS, and chapter 8 evaluated its effectiveness in refining TFS. Chapter 8 also showed that refinement could be used to carry out maintenance as well as refinement, and that TFS provided opportunities to

evaluate KRUST in both these roles. The upgrade from TFS-1A to TFS-1B was an example of debugging, since only minor changes were made. KRUST proved successful in identifying and fixing all the faults in TFS-1A, impressing the domain expert, and demonstrating its effectiveness in a debugging role.

The upgrade from TFS-1B to TFS-2 represented what the expert described as a paradigm shift in formulation policy, and originally required several man-months of work. This upgrade should be described therefore as maintenance, rather than debugging. KRUST was moderately successful in its attempts to perform this upgrade automatically. The expert judged that KRUST had identified the majority of the points where changes were needed. In some cases, the new rules generated by KRUST matched TFS-2's exactly; in others, they were reasonable alternatives.

Two conclusions were drawn from the application of KRUST to TFS-1B.

- Refinement of a design system is harder than refining a classificatory system, because there is often no one best answer. Consequently, a domain expert needs to assist in selecting good refinements.
- When maintenance is performed on a KBS, major changes are made to the rule base, and these may interact in complex ways. Often a number of changes will interact to produce a change in behaviour. When a refinement tool attempts to duplicate these changes, it may not be able to generate them all in a single run. Because of the interactions, implementation of some but not all the changes may not result in any observable improvement in the system's behaviour. Therefore a domain expert is again needed to identify good refinements, but for a different reason.

Another way to view the proposed interaction between domain expert and KRUST is to view KRUST as a tool to assist the expert in debugging or maintaining a system, rather than, as at present, a "black box" which performs the task automatically.

10.2 Secondary goal

A secondary goal of the project was to create a generic, extensible refinement tool, capable of refining a variety of different shells. This section examines the extent to which this was achieved.

Chapter 5 described that adaptations that had to be made to KRUST in order to apply it to PFES. Two conclusions can be drawn from this account.

- KRUST had a number of features that facilitated its adaptation.
- The current project caused KRUST to develop further in the direction of a truly generic tool.

The version of KRUST available at the start of this project had a number of properties which made it readily applicable to a variety of shells. These were its knowledge hierarchy, parser, and operator tool-sets. The hierarchy and tool-sets proved adaptable to the needs of PFES, but the parser did not. The parser facilitates the construction of a KBS translator given a set of grammar rules. However, the KRUST parser could not be applied to PFES rules, because the elements in a PFES rule object can appear in an arbitrary order. Instead, a separate translator had to be written.

It was noted in section 5.4.8 that the translation of PFES agendas was difficult, and required application-specific knowledge to recognise the relationship between successive agenda operations; for example, to recognise in what circumstances a value written to an agenda represented a property of another item on the agenda. However, the difficulty was caused by the flexible nature of the agenda structure, and the fact that the semantics of the agenda operations was often implicit; there is nothing in the PFES code to indicate explicitly the relationship between the items on an agenda. The conclusion from this is that refinement is easier for rules which have a declarative rather than a procedural behaviour, and that it may in some circumstances be possible to transform rules into a more declarative form by a change in representation.

KRUST's hierarchy, on the other hand, proved able to represent PFES rule elements with very little adaptation. The most distinctive PFES items, agendas, were able to be represented as ordered terms, and a small extension to the hierarchy permitted KRUST to handle PFES's use of Lisp functions. Moreover, ongoing work on applying KRUST to CLIPS and POWERMODEL shows that the hierarchy can readily be further extended to handle the complex rule-elements found in these two environments. Johnson (1997) confirms that different shells offer a wide variety of rule elements, but uses a primarily syntactically-based approach to construct a common framework, in contrast to KRUST's semantically-based approach.

KRUST's remaining generic feature is its collection of operators. The power of these operators is demonstrated by the fact that they were immediately applicable to PFES, once an internal representation of PFES rules could be constructed. In addition, KRUST possesses a lower-level toolset of functions which manipulate the data-structures used to represent rules and refinements. These are designed to be used in the creation of new operators, and indeed facilitated the creation of the new operators described in chapter 6. Examples of these lower-level functions are **replace_condition**, which replaces a given rule condition with a different one, and **add_condition**.

10.3 Traces give added value

The original version of KRUST obtained information about the behaviour of a KB by submitting queries. This is an appropriate method for exploring a KBS that can be regarded as a logical theory, where the query effectively re-runs part of the proof, and where the act of executing a query does not change the contents of the knowledge-base. However, none of these things is true of a forward-chaining system. If the query-based interface is applied to a forward-chaining system, it can provide information only about the final state of the system. It cannot take account of facts which may have been added to working memory and then removed, nor of the order in which actions were performed. A second disadvantage of the query-based interface is that it is only applicable to KBSs which can support external queries. Given a KBS such as PFES which does not offer this facility, an alternative approach is needed.

KRUST's use of execution traces to determine the behaviour of PFES solves both these problems. The use of traces has several advantages.

- It allows KRUST to refine non-monotonic systems.
- It provides information about the order of events, not just about what is true at the end of a run. In addition to providing the information necessary for blame allocation and refinement generation, it allows the order to place further constraints on the correct behaviour of a system. There is no point in the user stating that a KBS must perform actions in a certain order if KRUST is unable to determine the order in which they were performed. This facility does not appear to be useful in

the refinement of PFES, but it might be applicable when refining more interactive systems, such as those refined by CLIPS-R (Murphy & Pazzani 1994).

- It allows KRUST to refine KBSs which can not be queried, such as PFES. Most shells provide some form of trace of rule execution, but in some cases it is hard, if not impossible, to submit the kind of queries used by the original KRUST. PFES is an example of a shell to which such queries can not be sent. POWERMODEL is a shell to which it is possible but hard to establish the necessary software interface.

10.4 Future Work

10.4.1 Further developing a generic tool

Section 10.2 has shown that KRUST already fits the description of a generic refinement tool, applicable to a number of shells. However, there are a number of ways in which it could be improved.

KRUST has grown by adding functionality required by each of the KBS shells to which it has been applied. Admittedly, this has been done in an organised way, and new knowledge elements fitted into a standard framework. Nonetheless, the tool has become unwieldy, and would benefit from being broken down into modules. In this way the user could select only the modules he required, and then customise them to his particular application. This idea is the basis for the current KRUSTWorks project[1]. This project is a continuation of several projects, including the work described in this thesis, and aims to produce the KRUSTWorks toolkit. It is proposed that KRUSTWorks should operate as shown in figure 10.1. The user will provide a KBS consisting of a KB together with an interpreter. The user also supplies a grammar for parsing the KB. KRUSTWorks will then perform an analysis of the KB, determining properties such as the kinds of rule elements present, and the direction of rule chaining. Guided by the user, KRUSTWorks will then generate and customise modules to perform translation, together with all the standard refinement tasks of refinement generation, filtering, implementation, and evaluation. The resulting refinement tool appears in the large rectangle at the bottom of figure 10.1. It corresponds in many ways to the present KRUST, but will be smaller, since it does not

[1]Supported by EPSRC grant GR/L38387

contain unnecessary functionality, and it will be customised to the needs of a particular application.

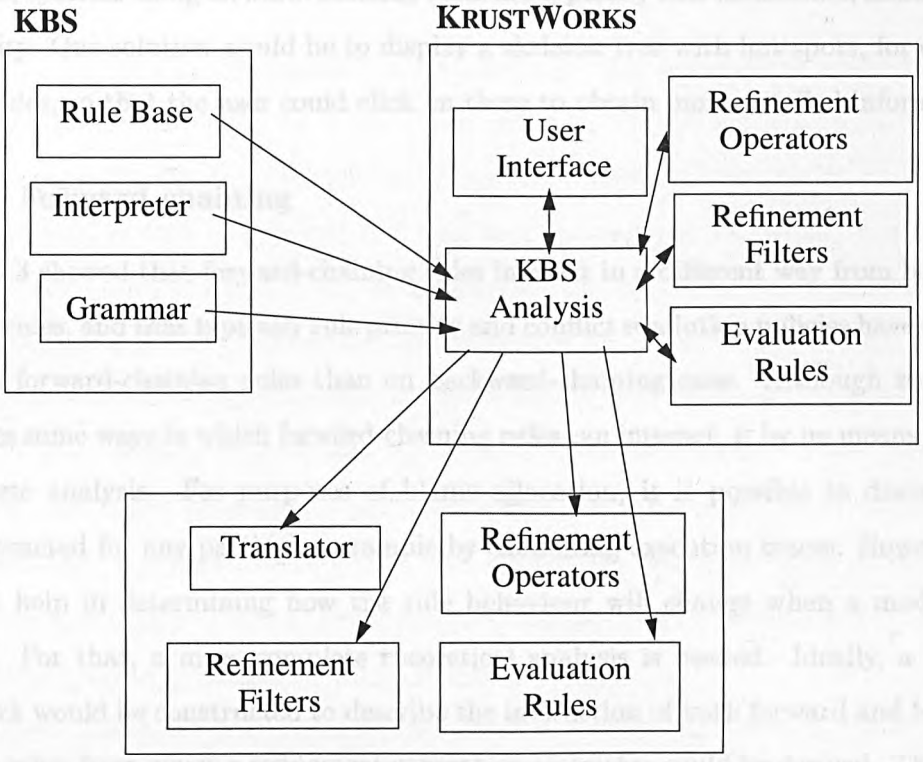

Figure 10.1: A generic refinement tool

10.4.2 User interface

A conclusion from the application of KRUST in a maintenance role is that there is a need for a domain expert to play a role in refinement filtering. In our experience with TFS, the expert was happy to assess individual new rules in isolation. However, in general it would be desirable to provide assistance to the expert by displaying the rule interactions in the form of a proof tree, making it clear how the proposed refinement is intended to affect the system's output. This would have been helpful on the occasion when the expert wrongly believed that the deletion of INSOLUBLE-FILLER-RULE was correct (section 8.6.9). The expert subsequently realised his mistake when he examined the behaviour of TFS-2, and realised the importance of the tablet property rule in filler selection. If the proof tree had been displayed, the role of the tablet property rule would have been clear.

Generating a graphical display for the behaviour of a design system is not a trivial

task, however. Frequently-used paradigms are "generate and test", and "generate and filter", which mean that many rules fire multiple times generating multiple conclusions. Moreover, systems using forward-chaining rules are typically non-monotonic, adding to the complexity. One solution would be to display a skeleton tree with hot-spots, for example, at the nodes, so that the user could click on these to obtain more detailed information.

10.4.3 Forward chaining

Section 5.3 showed that forward-chaining rules interact in a different way from backward-chaining rules, and that typically rule priority and conflict resolution policies have a greater affect on forward-chaining rules than on backward-chaining ones. Although section 5.3 illustrates some ways in which forward-chaining rules can interact, it by no means provides a complete analysis. For purposes of blame allocation, it is possible to discover how rules interacted for any particular example by examining execution traces. However, this does not help in determining how the rule behaviour will change when a modification is made. For that, a more complete theoretical analysis is needed. Ideally, a common framework would be constructed to describe the interaction of both forward and backward chaining rules, from which a refinement generation algorithm could be derived. This would simplify the implementation of the refinement generation module(s) in KRUSTWorks. In the absence of such a framework, it might be necessary to implement separate modules for backward and forward-chaining rules.

10.4.4 Refinement generation

Section 2.3.2 showed that extending the scope of a refinement tool from propositional to first-order logic greatly increases the potential complexity of the interaction between rules, and the way in which a faulty rule can lead to faulty behaviour in the system. It showed that there were faults which KRUST could not fix because the blame allocation procedure would fail to identify certain potentially faulty rules. The following example, slightly more concrete than those in section 2.3.2, illustrates the difficulty.

```
If   bathroom is steamy
Then temperature of water is 70.
```

```
If   temperature of water is ?t
     ?t < 60
Then turn on heater
```

Suppose the second rule is identified as a no-fire rule. For example, it might fail to fire because its second condition, `?t < 60`, is not satisfied. KRUST will try to fix the second rule by generalising the failed condition. However, KRUST will not realise that the first rule, which chains with the second and thus sets the value of `?t`, is also potentially at fault. If KRUST were to identify all potential causes for the second rule's failure, it would have to back-propagate the constraint `temperature of water < 60` through the proof tree, and identify all the possible rule-changes whereby that constraint could be satisfied.

KRUST does already possess this ability in a specialised form. When correcting a wrong-fire rule, it can propagate a particular value through a sequence of variable unifications and even arithmetic calculations, and identify the change at some ancestral node of the proof tree that will lead to the desired value being output (section 5.5.4). However, it can not propagate a constraint in this manner.

This section and the previous one have identified two weaknesses in KRUST's blame allocation and refinement generation strategies. A rigorous theoretical analysis of the ways in which forward and backward-chaining rules can interact would be a sensible starting point for devising more powerful algorithms.

10.4.5 Integration with Validation and Verification tools

Section 9.2.1 noted that recently-developed refinement tools have extended the definition of the *behaviour* of a system, where it is faults in the behaviour that trigger refinement. The only behaviour originally considered was the class to which a KBS assigned an example. Other behaviour which can be considered by KRUST and CLIPS-R are the various elements in a system's complex output, intermediate results generated by the system, and the order in which the system performs certain actions. One natural continuation of this development would be to derive input for the refinement process from other diagnostic tools. This would allow a refinement tool to become part of a KBS development toolkit. Various forms of interaction are possible. For example, a static analysis tool could be used to identify potentially faulty or missing rules, and a refinement tool could be used to fix

these faults if appropriate refinement examples were selected. Here, appropriate examples are those which would cause the faulty rule to fire, or which require the conclusion of the missing rule to be true. The use of a static analysis tool for example selection would complement the work of Palmer & Craw (1997), which uses similar reasoning to solve the problem of selecting appropriate judging examples for evaluating refined KBs. Other tools, such as SYCOJET (Ayel & Vignollet 1993) generate examples in an organised way. Such a tool would be an excellent source of training examples.

The idea of creating a toolkit containing a number of V&V tools, together with a refinement tool, was proposed as part of the ViVa project[2]. KRUST was the prototype improvement tool in ViVa (Craw 1996), but ViVa's refinement work finished when a project re-scoping concentrated effort on V&V issues.

10.5 Summary

The knowledge-refinement system KRUST has been applied to the tablet formulation system TFS, which solves a design problem. Design systems are significantly different from classificatory ones, and, until now, refinement tools have been applied only to classificatory systems.

The application of KRUST to TFS required modifications to KRUST's knowledge model, operators, and method of communication with the KBS. However, the adaptations of the knowledge model and operators were a matter of extension, not redesign, thus demonstrating that KRUST was to some extent a generic refinement tool.

The new method of communication derived information about the KBS's behaviour from execution traces rather than from direct queries. This conveyed the following advantages.

- It allowed KRUST to communicate with a system with a restricted software interface, whose working memory could not be accessed directly.
- It provided information about the order in which rules fired, which is particularly important for forward-chaining systems.
- It allowed KRUST to reason about a non-monotonic system.

[2]ESPRIT project 6125: **V**erification, **I**mprovement and **VA**lidation of Knowledge Based Systems

Once KRUST had been adapted to work with TFS, a number of inductive operators were implemented. These all take a similar approach, identifying features which distinguish faulty from non-faulty examples and traces, and using these features to modify rule conditions, or learn new ones. They permit KRUST to use information from multiple examples at an earlier stage than before, and so make the refinement process more efficient. The same approach has been applied with a different purpose; to provide additional information identifying rules which are not responsible for a fault, and so improve the effectiveness of refinement filtering.

Next, some applications of clustering to example selection at various stages of KRUST's refinement process were demonstrated, and these were later used in the experiments which evaluated KRUST's effectiveness.

The role of refinement in software development was then examined, and it was shown that refinement could be used in a maintenance role, as well as a debugging role. The refinement of TFS-1A was a debugging task, but the upgrading of TFS-1B to TFS-2 corresponded to a policy change in some aspects of the formulation process, so was a maintenance task. Experiments were performed to evaluate KRUST's effectiveness in both these tasks. KRUST proved highly effective in debugging, identifying and fixing all the faults in TFS-1A. KRUST was less effective in performing the harder maintenance task, but identified the majority of the areas where changes needed to be made, and in many cases generated reasonable alternatives to the changes which appeared in TFS-2. The principal reason for the difficulty lay in the nature of the design process, where there is no one best solution. KRUST would have benefited from expert assistance in selecting good refinements, since a simple comparison with the oracle's behaviour was too restrictive. In addition, expert advice on which attributes were likely to be relevant would have provided useful input to KRUST's induction of new rule conditions.

It has thus been shown that KRUST was effective at debugging an industrial design system. When used in a maintenance role, however, it should be seen as a tool to assist a human expert, rather than as a black-box which can perform the whole task unaided.

Appendix A

A typical KRUST run

This chapter illustrates the behaviour of KRUST for a typical run, using annotated output from the program. The run is taken from the experiments on the refinement of TFS-1A (section 8.5), so the role of system is taken by TFS-1A, and the role of oracle is taken by TFS-1B.

At one point in the iterative experimental procedure described in section 8.4 example number 9 is selected. The system and oracle outputs for this example are compared, and found to differ in a number of attribute values. The method of rule chaining described in section 5.2 is used to determine that the only independent attribute among the incorrect ones is the filler. KRUST therefore generates a query based on the difference between system and oracle values for filler.

```
example number is 9
query is ((== FORMULATION FILLER CALCIUM-CARBONATE)
          (== FORMULATION FILLER CALCIUM-PHOSPHATE)
          (== FORMULATION FILLER _FILLER))
```

This means that the system value for filler is calcium phosphate, and the oracle value is calcium carbonate. The third entry represents the query which is incorrectly answered by the system.

The next step is blame allocation, where KRUST classifies the system's end-rules according to their faulty behaviour (section 4.4.2). End-rules are those whose conclusion matches the above query; in other words, those which derive the filler for the formulation.

```
{{ Getting Tagged Rules }}

*TAGGED RULES*
WRONG_FIRE
RULE GET-INSOLUBLE-FILLER [FC 50]:
IF REQD-FILLER-SOLUBILITY of SPECIFICATION == INSOLUBLE
AND ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
AND SOLUBILITY of _FILLER == _SOL
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOL < (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REFINE-ATTRIBUTE(FORMULATION FILLER _FILLER)

NOCAN_FIRE
RULE GET-SOLUBLE-FILLER [FC 50]:
IF REQD-FILLER-SOLUBILITY of SPECIFICATION == NOT-INSOLUBLE
AND ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
AND SOLUBILITY of _FILLER == _SOL
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOL >= (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REFINE-ATTRIBUTE(FORMULATION FILLER _FILLER)

CAN_FIRE
RULE GET-ANY-FILLER [FC 0]:
IF ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
THEN REFINE-ATTRIBUTE(FORMULATION FILLER _FILLER)

CAN_FIRE
RULE SET-FILLER-LEVEL [FC -10]:
IF FILLER of FORMULATION == _FILLER
AND DRUG of FORMULATION == _DRUG
AND _DRUG of FORMULATION == _DOSE
AND TARGET-TABLET-WEIGHT of SPECIFICATION == _WEIGHT
AND DRUG+FILLER-CONCENTRATION of SPECIFICATION == _DRUG+FILLER-CONC
AND #<Arith: (/ _DOSE _WEIGHT)> -> _DRUG-CONC
AND #<Arith: (- _DRUG+FILLER-CONC _DRUG-CONC)> -> _FILLER-CONC
THEN _FILLER of FORMULATION = _FILLER-CONC
```

Note that the last of the tagged rules does not draw a conclusion about the filler, but about the filler concentration, so that it is not in fact an end-rule. However, it has been included by KRUST because its conclusion unifies with the query `FILLER of FORMULATION = _FILLER`. There is no way in which KRUST could recognise that such rules are irrelevant to the current query, unless it were provided with some form of meta-knowledge about the

structure of rules in the TFS KB.

The next step is refinement generation. KRUST does not provide a detailed trace at this stage, because it will display all the remaining refinements once they have undergone filtering. KRUST's trace therefore continues as follows, describing the filtering of the generated refinements.

```
126 fire-enabling refinements produced

{{ Filtering Refinements }}

{{ Applying filter "Superset" }}
Filter rank is 100.
49 items after "Superset", 1 iteration(s)

{{ Applying filter "Spec/Gen Conflict" }}
Filter rank is 80.
49 items after "Spec/Gen Conflict", 1 iteration(s)

{{ Applying filter "Weight Threshold" }}
Filter rank is 60.
Note: KB is unweighted - no metaknowledge filtering
49 items after "Weight Threshold", 1 iteration(s)

{{ Applying filter "Remove Most Complex" }}
Filter rank is 10.
Note: application unnecessary - filter's max output is 50
49 items after "Remove Most Complex", 0 iteration(s)

49 fire-enabling refinements after filtering

49 refinements in total for FILLER of FORMULATION == _FILLER

Refinement generation done
```

KRUST now lists the 49 surviving refinements. A typical selection appears below.

```
Experiment 1; To CORRECT WRONG_FIRE GET-INSOLUBLE-FILLER:
Specialise REQD-FILLER-SOLUBILITY of SPECIFICATION == INSOLUBLE in
RULE GET-INSOLUBLE-FILLER [FC 50]:
IF REQD-FILLER-SOLUBILITY of SPECIFICATION == INSOLUBLE
AND ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
AND SOLUBILITY of _FILLER == _SOL
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOL < (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REFINE-ATTRIBUTE(FORMULATION FILLER _FILLER)

Experiment 50; To ALLOW NOCAN_FIRE GET-SOLUBLE-FILLER:

To ENABLE NOCAN_FIRE GET-SOLUBLE-FILLER:

Move
RULE GET-INSOLUBLE-FILLER [FC 50]:
IF REQD-FILLER-SOLUBILITY of SPECIFICATION == INSOLUBLE
AND ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
AND SOLUBILITY of _FILLER == _SOL
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOL < (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REFINE-ATTRIBUTE(FORMULATION FILLER _FILLER)
below GET-SOLUBLE-FILLER
Where:
_FILLER = CALCIUM-CARBONATE
>> AND <<
Generalise _SOLUBILITY < (MIN-VAL _SLIGHTLY-SOLUBLE) in
RULE INSOLUBLE-DRUG-RULE [FC 0]:
IF DRUG of FORMULATION == _DRUG
AND SOLUBILITY of _DRUG == _SOLUBILITY
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOLUBILITY < (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REQD-FILLER-SOLUBILITY of SPECIFICATION = NOT-INSOLUBLE
Where:
_SLIGHTLY-SOLUBLE = (1 10)
_SOLUBILITY = 50.0

-->> OR <<--

Experiment 48; To ALLOW NOCAN_FIRE GET-SOLUBLE-FILLER:

To ENABLE NOCAN_FIRE GET-SOLUBLE-FILLER:

Move
RULE GET-INSOLUBLE-FILLER [FC 50]:
IF REQD-FILLER-SOLUBILITY of SPECIFICATION == INSOLUBLE
AND ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
```

```
AND SOLUBILITY of _FILLER == _SOL
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOL < (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REFINE-ATTRIBUTE(FORMULATION FILLER _FILLER)
below GET-SOLUBLE-FILLER
Where:
_FILLER = CALCIUM-CARBONATE
>> AND <<
Generalise _SOLUBILITY < (MIN-VAL _SLIGHTLY-SOLUBLE) in
RULE INSOLUBLE-DRUG-RULE [FC 0]:
IF DRUG of FORMULATION == _DRUG
AND SOLUBILITY of _DRUG == _SOLUBILITY
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOLUBILITY < (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REQD-FILLER-SOLUBILITY of SPECIFICATION = NOT-INSOLUBLE
Where:
_SLIGHTLY-SOLUBLE = (1 10)
_SOLUBILITY = 50.0

-->> OR <<--

Experiment 3; To CORRECT WRONG_FIRE GET-INSOLUBLE-FILLER:
Specialise SOLUBILITY of _DRUG == _SOLUBILITY in
RULE SOLUBLE-DRUG-RULE [FC 0]:
IF DRUG of FORMULATION == _DRUG
AND SOLUBILITY of _DRUG == _SOLUBILITY
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOLUBILITY >= (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REQD-FILLER-SOLUBILITY of SPECIFICATION = INSOLUBLE
Where:
_SOLUBILITY = 50.0
_DRUG = DRUG-A

-->> OR <<--

Experiment 6; To CORRECT WRONG_FIRE GET-INSOLUBLE-FILLER:
Specialise SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE in
RULE SOLUBLE-DRUG-RULE [FC 0]:
IF DRUG of FORMULATION == _DRUG
AND SOLUBILITY of _DRUG == _SOLUBILITY
AND SLIGHTLY-SOLUBLE of SOLUBILITY == _SLIGHTLY-SOLUBLE
AND _SOLUBILITY >= (MIN-VAL _SLIGHTLY-SOLUBLE)
THEN REQD-FILLER-SOLUBILITY of SPECIFICATION = INSOLUBLE
Where:
_SLIGHTLY-SOLUBLE = (1 10)

-->> OR <<--
```

```
Experiment 11; To CORRECT WRONG_FIRE GET-INSOLUBLE-FILLER:
Generalise MAX-LEVEL of _FILLER == _LEVEL in
RULE REMOVE-EXCESSIVE-FILLERS [FC 10]:
IF ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
AND MAX-LEVEL of _FILLER == _LEVEL
AND FILLER-CONCENTRATION of SPECIFICATION == _CONC
AND _CONC > _LEVEL
THEN REMOVE-FROM-AGENDA(FILLER-AGENDA RANK-FILLERS _FILLER)
Where:
_FILLER = CALCIUM-PHOSPHATE
```

KRUST now generates one or more refined KBs for each of these experiments. Many refinements can be implemented by more than one operator; for example, specialisation can be performed by deleting a rule, adjusting a threshold, or changing a comparison operator. This could cause the number of refined KBs generated to be greater than the number of experiments. On the other hand, two or more experiments may lead to identical KBs. This can happen if, for example, two experiments each attempt to specialise a different condition within the same rule. If neither condition can be specialised, each of the experiments will simply delete the rule, thus creating two identical KBs. Since duplicates are removed, this phenomenon would tend to reduce the number of refined KBs eventually returned. In the current instance, the 49 experiments lead to 53 implemented KBs, but 36 are duplicates, so 17 are returned.

```
Note: 36 duplicate KBs removed

17 refined KBs generated
```

At this stage, the refined KBs exist only in KRUST's internal representation. The next step is to write them to file as PFES KBs.

```
{{ Saving KB files }}
Writing /home/grad/rab/krust/tmp/vesp/nkb149_1
Writing /home/grad/rab/krust/tmp/vesp/nkb137_1
Writing /home/grad/rab/krust/tmp/vesp/nkb119_1
```

```
Writing /home/grad/rab/krust/tmp/vesp/nkb130_1
Writing /home/grad/rab/krust/tmp/vesp/nkb11_2
...
```

The 17 refined KBs are then run on the refinement case. To be accepted, they must generate the correct output value for the particular field that is the subject of the current query. Here, the field in question is filler. Those that give an incorrect value for this field are rejected by the refinement example filter.

The section of trace included below shows just two of the KBs being tested on the refinement example. For each example, KRUST display the input values, and any discrepancies between the KB's and the oracle's choice of filler.

```
{{ Applying filter "Refinement Case" }}
Filter rank is 100.
TFS-1A (1): DRUG-A Dose 210 mg Fillers 1
Atom space: 4331 (limit 2000)
Cons space: 4079 (limit 2000)

Extracting formulation from state file...
     Formulation-Filler-List
     (CALCIUM-PHOSPHATE 0.3427138)
     (CALCIUM-CARBONATE 0.3499289)

Result of test-example for nkb149_1
  error_pair_list ((0 1))

TFS-1A (1): DRUG-A Dose 210 mg Fillers 1
Atom space: 5586 (limit 2000)
Cons space: 5725 (limit 2000)

Extracting formulation from state file...

Result of test-example for nkb11_1
  error_pair_list ((0 0))
```

The trace shows that the first refined KB is still recommending calcium phosphate where the oracle recommends calcium carbonate. No discrepancy is displayed by the second KB, which indicates that the second KB recommends the correct filler. The variable

error_pair_list gives the number of errors in the specification and formulation respectively. Since only one field is being tested at this point, and it lies in the formulation, the error_pair_list will be ((0 1)) for KBs which fail, and ((0 0)) for KBs which succeed.

The atom and cons spaces mentioned in the trace refer to the memory used by PFES on the PC. When the space available drops below a certain threshold, KRUST will automatically cause the application to exit and re-start.

At this point, only one KB has survived the filtering process. This KB has the label nkb11_1, and corresponds to Experiment 11, listed earlier.

```
Experiment 11; To CORRECT WRONG_FIRE GET-INSOLUBLE-FILLER:
Generalise MAX-LEVEL of _FILLER == _LEVEL in
RULE REMOVE-EXCESSIVE-FILLERS [FC 10]:
IF ON-AGENDA(FILLER-AGENDA _TASK _FILLER)
AND MAX-LEVEL of _FILLER == _LEVEL
AND FILLER-CONCENTRATION of SPECIFICATION == _CONC
AND _CONC > _LEVEL
THEN REMOVE-FROM-AGENDA(FILLER-AGENDA RANK-FILLERS _FILLER)
Where:
_FILLER = CALCIUM-PHOSPHATE
```

The suffix "1" in nkb11_1 is necessary because a number of different KBs may be generated if more than one operator are applicable to experiment 11. The KB generated by a second generalisation operator would be labelled nkb11_2, and so on.

At this point, only one refined KB survives, to the remaining KB filters have no effect.

```
1 items after "Refinement Case", 1 iteration(s)

{{ Applying filter "Chestnut Cases" }}
Filter rank is 80.
Note: application unnecessary - filter's max output is 1
1 items after "Chestnut Cases", 0 iteration(s)

{{ Applying filter "Best Accuracy" }}
Filter rank is 60.
Note: application unnecessary - filter's max output is 1
```

```
1 items after "Best Accuracy", 0 iteration(s)

{{ Applying filter "Depth" }}
Filter rank is 40.
Note: application unnecessary - filter's max output is 1
1 items after "Depth", 0 iteration(s)

{{ Applying filter "Disruptiveness" }}
Filter rank is 20.
Note: application unnecessary - filter's max output is 1
1 items after "Disruptiveness", 0 iteration(s)


{{ Applying filter "Random" }}
Filter rank is 0.
Note: application unnecessary - filter's max output is 1
1 items after "Random", 0 iteration(s)
```

Although only one *refined* KB has survived, KRUST must still check that it is more accurate than the original KB, so it now runs both KBs on the judging examples. As before, KRUST displays any discrepancies between the output of the KB being evaluated and the oracle. It also calculates the total number of errors made by each KB over all the judging examples, and finally derives an error rate by dividing the number of errors by the total number of possible errors. At this stage, it is the overall accuracy of the KBs that is important, so all discrepancies between KB and oracle output are noted, not just those that concern the filler.

```
{{ Judging KBs }}
TFS-1A: DRUG-A Dose 160 mg Fillers 1
Converting DOS state file to Unix...
Formulation-Binder
     (GELATIN 0.041237112)
     (GELATIN 0.021052632)

TFS-1A: DRUG-A Dose 210 mg Fillers 1
Converting DOS state file to Unix...
Formulation-Binder
     (GELATIN 0.041237112)
```

```
     (GELATIN 0.021052632)

TFS-1A (1): DRUG-I Dose 210 mg Fillers 2
Converting DOS state file to Unix...
....
Formulation-Filler-List
     (LACTOSE 0.2347802)  (MAIZE-STARCH 0.10101011)
     (MAGNESIUM-CARBONATE 0.16036965)  (MAIZE-STARCH 0.17542066)
Formulation-Binder
     (PVP 0.030303031)
     (HPMC 0.030303031)
....
```

Having tested both the refined and the original KB on all the judging examples, KRUST displays the error rate for each, and returns the KB with the lower error rate. In this case, `nkb11_1` performs better than the original KB.

```
Function: ACCURACY
Ratings:
nkb11_1 has rating 0.48
tfs-1a has rating 0.5

The Refined KBs judged to be best:

nkb11_1:
ADD (RULE770)
Operators: "Add Fact"

{{ Refining Completed }}
```

The result of the run is that KRUST returns a single best refined KB. This KB was created by adding the fact

```
MAXIMUM-LEVEL of CALCIUM-PHOSPHATE = 0.3
```

to the TFS-1A database.

Bibliography

Alvey (1987). The PFES report, volume three: The formulations kernel, Logica UK Ltd.

Atkinson, L. & Hartley, P. (1983). *An introduction to numerical methods with Pascal*, Addison-Wesley.

Ayel, M. & Vignollet, L. (1993). SYCOJET and SACCO, two tools for verifying expert systems, *International Journal of Expert Systems: Research and Applications* **6**(3): 357–382.

Baffes, P. T. & Mooney, R. J. (1993). Symbolic revision of theories with M-of-N rules, *in* R. Bajcsy (ed.), *Proceedings of the Thirteenth IJCAI Conference*, Chambery, FRANCE, pp. 1135–1140.

Beckwith, R., Fellbaum, C., Gross, D. & Miller, G. (1991). WordNet: a lexical database organized on psycholinguistic principles, *in* U. Zernik (ed.), *Lexical Acquisition: Exploiting On-Line resources to build a lexicon*, Lawrence Erlbaum, pp. 211–232.

Boswell, R. A. (1986). Analytic goal-regression: Problems, solutions and enhancements, *in* L. Steels (ed.), *Proceedings of the ECAI86 Conference*, ECAI 86, pp. 43–54.

Boswell, R., Craw, S. & Rowe, R. (1996). Refinement of a product formulation expert system, *Proceedings of the ECAI-96 Workshop on Validation, Verification and Refinement of KBS*, Budapest, HUNGARY, pp. 74–79.

Boswell, R., Craw, S. & Rowe, R. (1997). Knowledge refinement for a design system, *in* E. Plaza & R. Benjamins (eds), *Proceedings of the European Knowledge Acquisition Workshop (EKAW97)*, Springer, Sant Feliu de Guixols, Spain, pp. 49–64.

Brunk, C. (1996). *An investigation of knowledge intensive approaches to concept learning and theory refinement*, PhD thesis, University of California, Irvine.

Brunk, C. & Pazzani, M. (1995). A lexically based semantic bias for theory revision, *in* A. Prieditis & S. Russell (eds), *Proceedings of the Twelfth International Conference on Machine Learning*, Morgan Kaufmann, Tahoe City, CA, pp. 81–89.

Bundy, A., Silver, B. & Plummer, D. (1985). An analytical comparison of some rule-learning programs, *Artificial Intelligence* **27**: 137–181.

Cain, T. (1991). The DUCTOR: A theory revision system for propositional domains, *Machine Learning: Proceedings of the Eighth International Workshop on Machine Learning*, pp. 485–489.

Clark, P. & Boswell, R. A. (1991). Rule induction with CN2: some recent improvements, *in* Y. Kodratoff (ed.), *Proceedings of the Fifth European Working Session on Learning*, Springer-Verlag, pp. 151–163.

Craven, M. W. & Shavlik, J. W. (1994). Using sampling and queries to extract rules from trained neural networks, *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 37–45.

Craw, S. (1991). *Automating the Refinement of Knowledge Based Systems*, PhD thesis, University of Aberdeen.

Craw, S. (1996). Refinement complements verification and validation, *International Journal of Human-Computer Studies* **44**(2): 245–256.

Craw, S., Boswell, R. & Rowe, R. (1997). Knowledge refinement to debug and maintain a tablet formulation system, *Proceedings of the 9TH IEEE International Conference on Tools with Artificial Intelligence (TAI'97)*, IEEE Press, Newport Beach, CA, pp. 446–453.

Craw, S. & Hutton, P. (1995). Protein folding: Symbolic refinement competes with neural networks, *in* A. Prieditis & S. Russell (eds), *Machine Learning: Proceedings of the Twelfth International Conference*, Morgan Kaufmann, Tahoe City, CA, pp. 133–141.

Craw, S. & Sleeman, D. (1990). Automating the refinement of knowledge-based systems, *in* L. C. Aiello (ed.), *Proceedings of the ECAI90 Conference*, Pitman, Stockholm, Sweden, pp. 167–172.

Craw, S. & Sleeman, D. (1991). The flexibility of speculative refinement, *in* L. Birnbaum & G. Collins (eds), *Machine Learning: Proceedings of the Eighth International Workshop*, Morgan Kaufmann, Evanston, IL, pp. 28–32.

Craw, S., Sleeman, D., Boswell, R. A. & Carbonara, L. (1994). Is knowledge refinement different from theory revision?, *in* S. Wrobel (ed.), *Proceedings of the MLNet Familiarization Workshop on Theory Revision and Restructuring in Machine Learning (ECML-94)*, GMD Technical Report Number 842, Catania, ITALY, pp. 32–34.

Davis, R. & Lenat, D. (1982). *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill.

Frank, J., Rupprecht, B. & Schmelmer, W. (1997). Knowledge-based assistance for the development of drugs, *IEEE Expert* **12**(1): 40–48.

Ginsberg, A. (1988a). *Automatic Refinement of Expert System Knowledge Bases*, Research Notes in Artificial Intelligence, Pitman, London.

Ginsberg, A. (1988b). Theory revision via prior operationalization, *Proceedings of the Sixth National Conference on Artificial Intelligence*, Minneapolis, MN, pp. 590–595.

Ginsberg, A. (1990). Theory reduction, theory revision, and retranslation, *Proceedings of the Eighth National Conference on Artificial Intelligence*, Cambridge, MA, pp. 777–782.

Girratano, J. & Riley, G. (1994). *Expert Systems: principles and programming*, 2 edn, PWS publishing company, Boston.

Johnson, V. M. (1997). Building a composite syntax for expert system shells, *IEEE Expert* **12**(6): 60–66.

Koppel, M., Segre, A. M. & Feldman, R. (1994). Getting the most from flawed theories, *Proceedings of the Eleventh International Conference on Machine Learning*, pp. 139–147.

Logica (1988). *Product formulation expert system: User Guide for the IBM PC*, Logica UK Ltd.

Mahoney, J. J. & Mooney, R. J. (1993). Combining connectionist and symbolic learning to refine certainty-factor rule-bases, *Connection Science (Special issue on Architecture for Integrating Neural and Symbolic Processing)* **5**: 339–364.

Merz, C. J. & Murphy, P. M. (1996). UCI Repository of machine learning databases (http://www.ics.uci.edu/~mlearn/MLRepository.html), Irvine, CA: University of California, Department of Information and Computer Science.

Michalski, R. & Stepp, R. (1990). Clustering, *in* S. Shapiro (ed.), *Encyclopaedia of Artificial Intelligence*, Vol. 1, Wiley, pp. 103–110.

Muggleton, S. (1987). Duce, an oracle based approach to constructive induction, *Proceedings of the Tenth IJCAI Conference*, Morgan Kaufmann, Milan, pp. 287–292.

Muggleton, S. (1992). Inductive logic programming, *Inductive logic Programming*, Academic Press, pp. 3–27.

Muggleton, S. & Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution, *Machine Learning: Proceedings of the Fifth International Conference*, Morgan Kaufmann, Ann Arbor, MI, USA, pp. 339–352.

Muggleton, S. & Feng, C. (1990). Efficient induction of logic programs, *Proceedings of the first conference on algorithmic learning theory*, Japanese Society for Artificial Intelligence, Ohmsha, Tokyo, pp. 368–381.

Murphy, P. M. & Pazzani, M. J. (1994). Revision of production system rule-bases, *in* W. W. Cohen & H. Hirsh (eds), *Proceedings of the Eleventh International Conference on Machine Learning*, Morgan Kaufmann, New Brunswick, NJ, pp. 199–207.

Ourston, D. & Mooney, R. (1994). Theory refinement combining analytical and empirical methods, *Artificial Intelligence* **66**: 273–309.

Palmer, G. (1995). Applying KRUST to a new KBS tool: experience with Kappa, *Technical Report 95/9*, SCMS, Robert Gordon University.

Palmer, G. J. & Craw, S. (1995). Utilising explanation to assist the refinement of knowledge-based systems, *Proceedings of the 3rd European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-95)*, pp. 201–211.

Palmer, G. J. & Craw, S. (1996). The role of test cases in automated knowledge refinement, *Research and Development in Expert Systems XIII: Proceedings of Expert Systems 96, 16th Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems*, SGES Publications, Cambridge, UK, pp. 75 – 90.

Palmer, G. J. & Craw, S. (1997). The selection of training cases for automated knowledge refinement, *in* J. Vanthienen & F. van Harmelen (eds), *Proceedings of the 4th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV97)*, Leuven, Belgium, pp. 205–215.

Pazzani, M. J. (1993). Student loan relational domain, In UCI Repository of Machine Learning Databases (Merz & Murphy 1996).

Pazzani, M. J. & Kibler, D. (1990). The utility of knowledge in inductive learning, *Technical Report 90-18*, University of California, Irvine, Department of Information and Computer Science.

Plotkin, G. (1971). *Automatic methods of inductive inference*, PhD thesis, Edinburgh University.

Quinlan, J. R. (1986). Induction of decision trees, *Machine Learning* **1**: 81–106.

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

Rasumssen, E. (1992). Clustering algorithms, *in* W. B. Frakes & R. Baeza-Yates (eds), *Information Retrieval: Data Structures and Algorithms*, Prentice Hall, London, pp. 419–442.

Richards, B. L. & Mooney, R. J. (1995). Refinement of first-order horn-clause domain theories, *Machine Learning* **19**(2): 95–131.

Rowe, R. (1993). An expert system for the formulation of pharmaceutical tablets, *Manufacturing Intelligence* (14): 13–15.

Sammut, C. & Bannerji, R. (1986). Learning concepts by asking questions, *Machine Learning – An Artificial Intelligence Approach*, Vol. 2, Morgan Kaufmann, pp. 167–192.

Towell, G. G., Shavlik, J. W. & Noordewier, M. O. (1990). Refinement of approximate domain theories by knowledge-based neural networks, *Proceedings of the Eighth National Conference on Artificial Intelligence*, MIT Press, Boston, MA, pp. 861–866.

Turner, J. (1991). Product formulation expert system, *Manufacturing Intelligence* (8): 12–14.

Wilkins, D. C. (1990). Knowledge base refinement as improving an incorrect and incomplete domain theory, *in* Y. Kodratoff & R. S. Michalski (eds), *Machine Learning Volume III*, Morgan Kaufmann, San Mateo, CA, pp. 493–513.

Wilkins, D. C., Clancey, W. J. & Buchanan, B. G. (1986). Overview of the ODYSSEUS learning apprentice, *in* T. M. Mitchell, J. G. Carbonell & R. S. Michalski (eds), *Machine Learning: A Guide to Current Research*, Kluwer, Boston, pp. 369–373.

Wilkins, D. C. & Tan, K.-W. (1989). Knowledge base refinement as improving an incorrect, inconsistent and incomplete domain theory, *in* B. Spatz (ed.), *Machine Learning: Proceedings of the Sixth International Workshop*, Morgan Kauffman, Ithaca, NY, pp. 332–337.

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

Wogulis, J. (1994). *An approach to repairing and evaluating first-order theories containing multiple concepts and negation*, PhD thesis, University of California, Irvine.

Wogulis, J. & Pazzani, M. (1993). A methodology for evaluating theory revision systems: Results with AUDREY II, *in* R. Bajcsy (ed.), *Proceedings of the Thirteenth IJCAI Conference*, Chambery, FRANCE, pp. 1128–1134.

Zelle, J. M. & Mooney, R. J. (1993). Combining FOIL and EBG to speed-up logic programs, *in* R. Bajcsy (ed.), *Proceedings of the Thirteenth IJCAI Conference*, Chambery, FRANCE, pp. 1106–1111.

Published Papers

- BOSWELL, CRAW and ROWE (1996). Refinement of a product formulation expert system, *Proceedings of the ECAI-96 Workshop on Validation, Verification and Refinement of KBS*, Budapest, Hungary, pp 74-79.

- BOSWELL, CRAW and ROWE (1997). Knowledge refinement for a design system, *Proceedings of the European Knowledge Acquisition Workshop (EKAW97)*, Springer, Sant Feliu de Guixols, Spain, pp 49-64.

- CRAW, BOSWELL and ROWE (1997). Knowledge refinement to debug and maintain a tablet formulation system, *Proceedings of the 9TH IEEE International Conference on Tools with Artificial Intelligence, (TAI97)*, IEEE Press, Newport Beach CA, pp 446-453.

Refinement of a Product Formulation Expert System

Robin Boswell[1], Susan Craw[1] and Ray Rowe[2]

Abstract. We describe extensions made to the knowledge refinement tool KRUST with the aim of developing a generic refinement tool, applicable to a variety of expert system shells. In particular, we describe the PFES shell and the tablet formulation application TFS written in PFES, discuss some of the differences between TFS and other systems, and show how we have extended KRUST in order to apply it to TFS.

1 Introduction

Knowledge refinement is the process of correcting errors in a KBS's rule base, triggered when test cases are wrongly solved by the KBS. This paper describes how the knowledge refinement tool KRUST has been applied to the product formulation shell PFES, a shell which differs in a number of significant ways from the backward-chaining diagnostic shells which have typically been the target both of KRUST and most other refinement tools.

We first describe the expert system shell PFES (§2), and the particular PFES application called TFS which we are using as a test-bed for KRUST. We go on to describe KRUST and how it has been extended to handle new types of knowledge-bases, and in particular how it has been applied to PFES (§3). We then describe the results so far achieved by KRUST (§4). Finally, we compare KRUST with other systems (§5).

2 The Application

2.1 The PFES expert system shell

PFES (Product Formulation Expert System) is a LISP expert system shell [9]. As the name suggests, the shell is designed for the formulation or synthesis of a solution to a problem, so it differs in some ways from shells designed for analytic tasks such as diagnosis. In particular, its control structure is task-based, and corresponding to each task is a rule-set which is executed by forward-chaining.

2.2 TFS

The tablet formulation expert system (TFS) solves the problem of selecting the inert substances, or *excipients*, which are needed to process a drug into a tablet [8]. When running TFS, the user first provides the name of a drug and its dosage, together with certain other desired properties of the final tablet; then TFS calculates a *formulation* consisting of the most appropriate material from each excipient type, and the quantity of each required. During the initial stages of this calculation, TFS also calculates some intermediate results called the *specification*; these are necessary properties of the formulation which follow directly from the user's requirements. TFS input is thus drawn both interactively from the user, and from databases containing chemical properties of drugs and excipients. TFS's output consists of the specification followed by the formulation for the desired tablet.

For our purposes, PFES/TFS has been modified so that it reads its requirements from file rather than interactively, and writes its output (specification and formulation) to file. Each requirement / specification / formulation triple is then regarded as an *example* by the refinement program. Figure 1 shows a typical example.

2.3 The Refinement Task

In general, the input to a refinement task is a KBS together with a set of examples, some or all of which are wrongly solved by the KBS. The refinement task then consists of correcting the KBS so that it correctly solves as many of the examples as possible. However on this occasion the task was presented in a slightly different form. Two versions of the KBS were supplied, TFS-1A and TFS-1B, where TFS-1B was a more recent version which fixed some bugs that had been detected in TFS-1A. Thus we use TFS-1B as an oracle to determine whether or not the output from TFS-1A is correct for any particular input. Consequently, our first task was to generate a set of TFS inputs, evenly distributed over the requirement space. This space is defined by three variables: Drug (13 values), number of fillers (2 values) and dosage (an integer, taking values between 1 and 360). Our generation routine selected all possible values of the first two variables, and an evenly-distributed subset of the possible dosage values starting at 10mg with a step size of 50mg. This gave a total of 182 different inputs. These were then passed both to TFS-1A and to the oracle (TFS-1B). The outputs from the two systems differed for the majority of examples.

TFS-1A's errors can be clustered (currently, by hand) into two groups, the examples in each group sharing similar symptoms. Examples from the first group show a binder concentration roughly twice the correct value, and examples from the second show calcium phosphate wrongly chosen as a filler.

[1] SCMS, Robert Gordon University, Aberdeen AB25 1HG. rab,smc@scms.rgu.ac.uk
[2] ZENECA Pharmaceuticals, Hurdsfield Industrial Estate, Macclesfield, Cheshire SK10 2NA

© 1996 R. Boswell, S. Craw & R. Rowe
Appears in *Proceedings of the ECAI-96 Workshop on Validation, Verification and Refinement of KBS*, pp74–79

TFS Input

Requirement:
- Drug: Drug-A
- Dose: 60 mg
- No of fillers: 2

TFS Output

Specification:
- full-stability: Yes
- drug-filler–concentration: 0.9
- minimum-tablet-weight: 100mg
- maximum-tablet-weight: 800mg
- target-tablet-weight: 260mg
- start-strategy: strategy-A
- filler-concentration: 66.9%
- typical-disintegrant: Maize-starch
- disintegrant-concentration: 0.05
- tablet-weight: 252.2mg
- total-concentration: 97%
- tablet-diameter: 8.73mm
- *... various other properties...*

Formulation:
- Tablet weight: 250mg
- Fillers: Lactose 66.7%, Calcium phosphate 2.4%
- Binder: Gelatin 4.1%
- Lubricant: Magnesium stearate 1.0%
- Disintegrant: Croscarmelose 2.1%

Figure 1. TFS input and output

Figure 2 shows two examples of each type of error. For examples 1 and 42, the system recommends a greater proportion of binder than the oracle, and for examples 5 and 33, the system recommends calcium phosphate as a filler while the oracle recommends calcium carbonate and magnesium carbonate respectively.

Incorrect quantity of binder

Example 1 — (10 mg, DRUG-A, 1 filler)
1A *System binder choice:* Gelatin 4.1%
1B *Oracle binder choice:* Gelatin 2.1%

Example 42 — (310 mg, DRUG-F, 2 fillers)
1A *System binder choice:* PVP 3.9%
1B *Oracle binder choice:* PVP 2.0%

Incorrect filler

Example 5 — (110 mg, DRUG-A, 1 filler)
1A *System filler choice:* Calcium phosphate 57.3%
1B *Oracle filler choice:* Calcium carbonate 58.5%

Example 33 — (110 mg, DRUG-F, 1 filler)
1A *System filler choice:* Calcium phosphate 55.5%
1B *Oracle filler choice:* Magnesium carbonate 56.6%

Figure 2. Formulations generated by TFS-1A and TFS-1B

3 KRUST and its application to PFES

3.1 The Operation of KRUST

The operation of any refinement system may be broken down into the following three tasks. (A more detailed account of the operation of KRUST may be found in [1]).

Blame allocation determines which rules or parts of rules might be responsible for the erroneous behaviour. In KRUST at this stage *no-fire rules* are identified, which produce the correct conclusion but did not fire, and *error-causing* rules which did (or could) fire, but do not produce the correct conclusion.

Refinement generation suggests rule modifications that may correct the erroneous behaviour. In general, there will be more than one way of fixing any particular error. In KRUST, refinement generation is based on the preceding blame allocation; no-fire rules may be enabled, e.g. by generalising a condition, and error-causing rules may be disabled, e.g. by specialising a condition. The chaining of rules must be considered, since it may not be the error-causing rule itself that is faulty, but rather one of the rules that caused it to fire.

Refinement selection picks the best of the possible refinements according to some criteria.

3.2 Applying KRUST to different knowledge bases

In this section we consider a number of issues that must be addressed in the development of a generic refinement tool, discussing first the general concepts, and then the particular requirements of PFES.

3.2.1 Rule element representation

Although it may at first appear that there is a wide variety in the representations used by various KBS development tools, there are only a limited number of roles that a rule element (condition or conclusion) can play within a rule [6]. For example, a condition can succeed or fail, bind variables, or be involved in rule chaining. These roles are the basis of KRUST's hierarchy of rule element types (figure 3): further types are likely to be added as new KBS tools require.

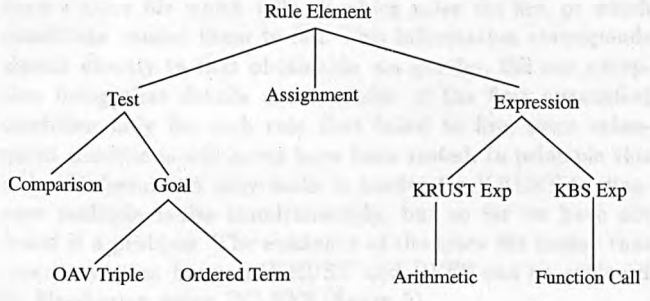

Figure 3. KRUST's hierarchy of rule elements

A *test* is a rule condition that can succeed or fail. Tests are currently of two types: comparisons (such as inequalities

or identity tests) and goals. A rule element is a *goal* if and only if it can be involved in rule chaining. Two types of goal have so far been defined: *oav_triples* (terms of the form "Attribute of Object is Value") and *ordered_terms*, corresponding to OPS5 tuples. *Expressions* are rule elements that return a value. This class is divided into *KRUSTExp*, which can be evaluated within KRUST and *KBSExp*, which have to be passed back to the KBS for evaluation.

3.2.2 Representation of PFES rules

Many of PFES's rule elements are standard, and therefore correspond directly with classes found in the hierarchy shown in figure 3; for example OAV triples, arithmetic calculations, and comparisons between variables. However, there is a group of rule elements that appear not to be described by the hierarchy: operations on *agendas*. PFES agendas are untyped lists; an item can be read and written to the top or bottom of an agenda, or directly below another given item.

However, TFS agendas can be interpreted as a mechanism for storing attribute-value data. Not all agendas have the same semantics, but the number of different possibilities actually employed within TFS is fairly limited. Two of the most common examples are shown in figure 4. Each example shows the contents of an agenda at some point during the running of TFS-1A, together with the rule elements that write to and read from the agenda. The Filler-Agenda is simply a list of excipients; their presence on the agenda indicates that they have passed a stability test. The Property-Agenda again shows a list of excipients, but now each excipient has an associated floating-point number, representing the value of a mechanical property. In both cases, the rule elements which read and write the agenda items can be represented as ordered terms, as follows.

Example 1

```
ON-AGENDA(FILLER-AGENDA,
          RANK-FILLERS, <FILLER>)
```

Example 2

```
ON-AGENDA(PROPERTY-AGENDA,
          RANK-FILLERS, <FILLER>)
AGENDA-UNLABELLED-ATTRIBUTE(PROPERTY-AGENDA,
          RANK-FILLERS, <FILLER>, <STABILITY>)
```

It will be noted that the PFES command

```
ADD <ITEM> TO-BOTTOM-OF <AGENDA>
```

has different translations in the two examples. Fortunately it is possible determine the correct translation from the context, both in these two cases and in other more complex situations which also arise in TFS.

3.2.3 Communication

During refinement, KRUST requires answers to the following queries:

- Can rule R be satisfied, and what variable bindings result?

[3] Terms in angle-brackets <> are PFES variables.

Snapshot of FILLER-AGENDA

```
MICROCRYSTALLINE-CELLULOSE
LACTOSE
CALCIUM-CARBONATE
CALCIUM-PHOSPHATE
```

Read/Write Operations on FILLER-AGENDA

Rule conclusion that adds an item to the agenda:

```
ADD <FILLER> TO-BOTTOM-OF FILLER-AGENDA
```

[3]

Rule condition that reads an item from the agenda:

```
READ <FILLER> FROM FILLER-AGENDA
```

Snapshot of PROPERTY-AGENDA

```
MICROCRYSTALLINE-CELLULOSE
46.2
LACTOSE
158.8
CALCIUM-CARBONATE
851.1
```

Read/Write Operations on PROPERTY AGENDA

Rule conclusions that add items to this agenda:

```
ADD <FILLER> TO-BOTTOM-OF PROPERTY-AGENDA
ADD <STABILITY> TO-BOTTOM-OF
    PROPERTY-AGENDA
```

Rule conditions that read items from this agenda:

```
READ <FILLER> FROM PROPERTY-AGENDA
READ <STABILITY> THE-ITEM-AFTER <FILLER>
    ON PROPERTY-AGENDA
```

Figure 4. Two agendas, and the PFES statements that read and write them

- If rule R cannot be satisfied, which condition(s) cause it to fail?

Originally this was achieved by KRUST communicating with the KBS directly. PFES does not allow this, since it must always answer complete formulation tasks, and it runs on a different platform from KRUST. However, PFES does produce a trace file which tells us which rules *did* fire, or which conditions caused them to fail. This information corresponds almost exactly to that obtainable via queries; the one exception being that details are available of the first unsatisfied condition only for each rule that failed to fire, since subsequent conditions will never have been tested. In principle this lack of information may make it harder for KRUST to diagnose multiple faults simultaneously, but so far we have not found it a problem. The existence of the trace file means that communication between KRUST and PFES can be achieved by file-sharing using PC-NFS (figure 5).

3.2.4 Chaining Direction

Unlike the backward-chaining systems originally handled by KRUST, PFES is a task-based system where each task corresponds to a rule-set, and the rules within each set are executed

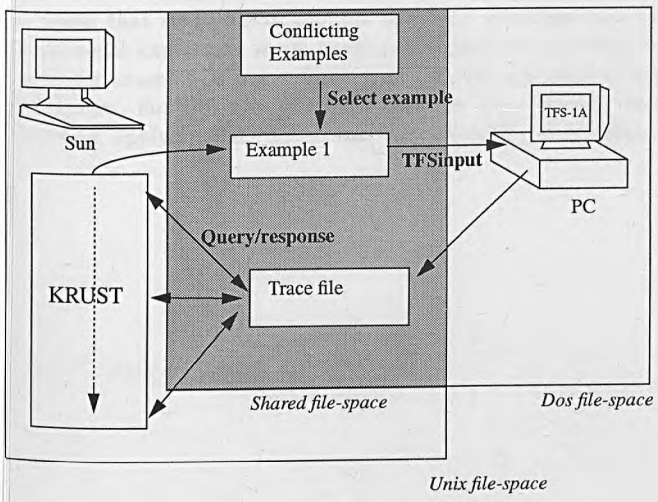

Figure 5. Interaction between Sun and PC during the running of KRUST

in forward-chaining mode. However, since KRUST's analysis of a KBS's behaviour is goal-driven, KRUST will work backwards from the conclusion to the initial facts regardless of the order in which the rules were originally executed. For example, suppose a rule base contains the fact A, Rule 1: $A \rightarrow B$ and Rule 2: $B \rightarrow C_{-}$, where C_{-} is an erroneous conclusion. Under forward-chaining, rule 1 will fire first, then rule 2. However, the trigger for refinement is the conclusion C_{-}, since that is the only aspect of the system's behaviour initially known to be wrong. The blame allocation process will then work *backwards* from C_{-} to rule 2 and then rule 1.

Furthermore, because rule chaining is mediated via the setting and reading of attribute values, and these are independent of PFES tasks, KRUST is also able to consider rules and rule-chaining regardless of task-set. For example, a rule in task `CHOOSE-FILLER` includes the action:

```
SET-ATTRIBUTE-VALUE
  ACCEPTABLE-STABILITY TO <TARGET-STABILITY>
```

and another rule in task `APPLY-FILLER-STRATEGY` includes the corresponding condition:

```
ATTRIBUTE-HAS-VALUE
    ACCEPTABLE-STABILITY <MIN-STABILITY>
```

The allocation of these two rules to different tasks means that the second rule will not be able to fire until task `APPLY-FILLER-STRATEGY` has been invoked, which may happen before or after task `CHOOSE-FILLER` is completed. Nonetheless, once the second rule *is* given a chance to fire, it is able to read the value written previously by the first rule. Therefore, for refinement, the first rule may be regarded as chaining with the second.

3.2.5 Blame Allocation with Variables

For propositional KBSs, KRUST identifies two rule classes: *error-causing* – rules which fired and gave the wrong conclusion, and *no-fire* – rules which gave the right conclusion but did not fire. KRUST has been extended to handle variables by adding a new class: *wrong-fire* – rules which fired and gave the wrong conclusion, but would have given the right conclusion if they had fired with a different set of variable bindings. In other words, wrong-fire rules are error-causing for one set of bindings and no-fire for another. Refinement generation has been extended to generate two types of refinement for wrong-fire rules: refinements to disable the wrong-fire rule, and refinements to correct the value assigned to an incorrectly-bound variable. In both cases the process is recursive, applying both to the wrong-fire rule and its antecedents. An example of refinement by value correction appears in the next section, on reasoning about arithmetic.

3.2.6 Arithmetic

We have extended KRUST so that it follows causal chains that include one or more arithmetic expressions. We illustrate this with reference to Example 1 in figure 2, for which the correct proportion of gelatin in the formulation is 2.1%, but TFS-1A recommends 4.1%.
The chain of rules leading to this conclusion includes:

```
(RULE SET-BINDER-CONC
  (CONDITIONS
      ...
    <ADJUSTED-CONC> := <CONC> / <TOTAL-CONC>
      ...
  (ACTIONS
    (ADD <ADJUSTED-CONC> TO-BOTTOM-OF
                         TABLET-REPORT-AGENDA)
)
```

In this rule, the erroneous value for the concentration of gelatin is assigned to the variable `<ADJUSTED-CONC>`, and placed on the `TABLET-REPORT-AGENDA`.

Previous versions of KRUST would simply have recommended specialising one of the rule's conditions in order to prevent the rule firing with the erroneous binding of `<ADJUSTED-CONC>`. However, KRUST now also reasons as follows. The incorrect value of the variable on the left-hand side of the assignment (`<ADJUSTED-CONC>`) could result from incorrect values for any of the variables on the right-hand side, and so could be corrected by modifying any one of those values leaving the others unchanged. Each modified value is obtained by solving the appropriate equation; here, for example

$$\texttt{<CONC>}_{\text{modified}} \;/\; \texttt{<TOTAL-CONC>}_{\text{existing}} = \texttt{<ADJUSTED-CONC>}_{\text{correct}}$$

and similarly for $\texttt{<TOTAL-CONC>}_{\text{modified}}$.

The point at which the recursive correction procedure terminates in this particular example is at the conclusion of an earlier rule, in which a value for concentration is hardwired:

```
(SET-ATTRIBUTE-VALUE
   FORMULATION <BINDER> 0.04)
```

The procedure then generates an appropriate refinement of the value `0.04` where it appears in this conclusion. This technique is not PFES-specific, and will apply to any KBS that uses arithmetic.

Note that refinements are not currently generated for the arithmetic expression itself, because the guidance provided by a single example is not sufficient to prevent a combinatorial explosion. For this and other reasons, we are currently considering applying KRUST to multiple examples in parallel.

3.2.7 PFES's many-valued output

One consequence of PFES's formulation task is that its output is a compound answer (figure 1), in contrast to the single result typically output from a diagnostic system. Our testing of TFS-1A revealed that its output typically differs from the correct values at only one or two points, but some examples can have as many as 12 points of difference. One approach has combined the refinements to correct individual errors. An alternative approach is to determine dependencies between faults, and attempt to fix the earliest fault(s) in the dependency chain first, in the hope that this will fix the later faults as well. A dependency chain linking attributes a_1 and a_2 is defined to be a sequence of rules $R_1, R_2, \ldots, R_n$ where R_1 includes a condition referring to the value a_1, R_n includes a conclusion setting the value of a_2 and for each pair R_i, R_{i+i} the conclusion of R_i matches a condition of R_{i+1}. The existence of such a chain indicates that the value of a_1 *may* affect the value of a_2. This technique of using dependency to select a potential prior cause out of set of faults has been applied (by hand, to date) to those few TFS-1A examples which exhibit large numbers of errors. In these cases, it has been possible to obtain either one or two values that are prior to all the others in terms of dependency.

4 Experiments

The extension of KRUST to handle the requirements of PFES has covered the blame allocation and refinement generation modules of KRUST. Thus, KRUST currently generates refinements, but does not test them. KRUST generated 93 possible refinements for example 1 in figure 2 and 61 refinements for example 5. Typical refinements for example 1 appear below.

```
Experiment 1;

To correct WRONG_FIRE UPDATE-FORMULATION:

Specialise ON-AGENDA(TABLET-REPORT-AGENDA
                     <TASK> <COMPONENT>)

in RULE UPDATE-FORMULATION:

IF ON-AGENDA(TABLET-REPORT-AGENDA
             <TASK> <COMPONENT>)
AND AGENDA-UNLABELLED-ATTRIBUTE
       (TABLET-REPORT-AGENDA
        <TASK> <COMPONENT> <CONC> 1)
AND CHECK_LEAF of <COMPONENT> == COMPONENT

THEN <COMPONENT> of FORMULATION = <CONC>

Experiment 8;

To correct WRONG_FIRE UPDATE-FORMULATION:

Specialise
   <WEIGHT> :=
      <ADJUSTED-CONC> * <TABLET-WEIGHT>

in RULE SET-BINDER-CONC1:

IF BINDER of FORMULATION == <EXCIPIENT>
AND <EXCIPIENT> of FORMULATION == <CONC>
AND TABLET-WEIGHT of FORMULATION
       == <TABLET-WEIGHT>
AND TOTAL-CONCENTRATION of SPECIFICATION
       == <TOTAL-CONC>
AND <ADJUSTED-CONC> := <CONC> / <TOTAL-CONC>
AND <WEIGHT> :=
       <ADJUSTED-CONC> * <TABLET-WEIGHT>

THEN ON-AGENDA(TABLET-REPORT-AGENDA
               EVALUATE-TABLET <EXCIPIENT>)

Experiment 18;

To correct WRONG_FIRE DEFAULT-BINDER-LEVEL:

Correct <BINDER> of FORMULATION = 0.04

in RULE DEFAULT-BINDER-LEVEL:

IF BINDER of FORMULATION == <BINDER>
THEN <BINDER> of FORMULATION = 0.04
```

Our next step will be to adapt the remaining KRUST modules to the requirements of PFES, so that the refinements we have generated can be filtered, implemented and tested.

5 Other work

Systems which correct faulty knowledge bases may be divided into two classes: knowledge base refinement, which includes KRUST, and theory revision [2]. The principal distinguishing features of the two classes are shown in table 1.

This comparison shows that, in general, knowledge refinement systems tend to be more applicable to real-world expert systems, because they take account of control knowledge and require fewer examples. Theory revision systems are more

Table 1. A comparison of Theory Revision and KB Refinement

Theory Revision	*KB Refinement*
Representation Propositional calculus, FOPL	Propositional calculus and production rules
ML technique Inductive (many examples)	EBL-like (few examples)
Knowledge sources A single KB	Many independent sources
Control strategies Prolog interpreter	Various: backward-chaining, forward-chaining, meta-rules ...
Refinement operators Adding/deleting rules/conditions	Can also specialise or generalise individual conditions

adept at adding new knowledge based on many examples. Therefore by developing KRUST into a generic refinement tool, applicable to a variety of different knowledge representations and control strategies, we are building on the existing strengths of knowledge-refinement systems in general. We now discuss in more detail features of KRUST which we believe are particularly significant.

- It refines shells having a variety of control strategies. Some other systems can reason about a shell's control strategy (e.g., Odysseus [10] makes use of meta-rules to guide the learning of new object rules) but they are often tied to a particular shell, so are not generally applicable (e.g., Odysseus refines Minerva KBSs).
- It can be applied to a wide range of shells. Many Theory Revision systems such as EITHER [5] and FORTE [7] are restricted to Prolog programs, and SEEK [4], Teiresias [3] and Odysseus are each applicable only to a single shell.
- It generates and tests many refined knowledge-bases. This feature is unique to KRUST. Other systems select refinements before implementing them as new KBSs, so may not detect unintended side-effects.

6 Conclusions

We have shown how KRUST can be applied to a shell having a number of peculiarities which cause it to differ significantly from the other shells to which KRUST has already been applied. We have solved two independent communication problems, both of which will be applicable to a number of expert systems: communicating with a PC-based shell; and generating refinements from the evidence in a trace. We have shown that even esoteric data-structures may be represented in a straightforward class hierarchy of rule element types. KRUST has been used to refine a shell whose control structures are different from the backward-chaining reasoning of Prolog, to which it was originally applied. Finally, we have extended the class of refinements which it is able to generate, and thus the number of faults which it is potentially able to fix.

ACKNOWLEDGEMENTS

We are grateful to Paul Bentley of Logica Cambridge Ltd. for his assistance with PFES.

[1] S. Craw and P. Hutton. Protein folding: Symbolic refinement competes with neural networks. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 133–141, Tahoe City, CA, 1995. Morgan Kaufmann.

[2] S. Craw, D. Sleeman, R. Boswell, and L. Carbonara. Is knowledge refinement different from theory revision? In S. Wrobel, editor, *Proceedings of the MLNet Familiarization Workshop on Theory Revision and Restructuring in Machine Learning (ECML-94)*, pages 32–34, Catania, ITALY, 1994. GMD Technical Report Number 842.

[3] R. Davis and D. Lenat. *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill, 1982.

[4] A. Ginsberg, S. M. Weiss, and P. G. Politakis. Seek2: A generalized approach to automatic knowledge base refinement. In *Proceedings of the Ninth IJCAI Conference*, pages 367–374, 1985.

[5] D. Ourston and R. Mooney. Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309, 1994.

[6] G. Palmer. Towards an extensible knowledge refinement tool. Technical Report 96/1, SCMS, Robert Gordon University, 1996.

[7] B. L. Richards and R. J. Mooney. Refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.

[8] R. Rowe. An expert system for the formulation of pharmaceutical tablets. *Manufacturing Intelligence*, 14:13–15, 1993.

[9] J. Turner. Product formulation expert system. *Manufacturing Intelligence*, (8):12 – 14, 1991.

[10] D. C. Wilkins. Knowledge base refinement as improving an incorrect and incomplete domain theory. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning Volume III*, pages 493–513. Morgan Kaufmann, San Mateo, CA, 1990.

Appears in *Proceedings of the 10TH European Knowledge Acquisition Workshop*, pp49–64, Springer

Knowledge Refinement for a Design System

Robin Boswell[1], Susan Craw[1] and Ray Rowe[2]

[1] School of Computer and Mathematical Sciences,
The Robert Gordon University, Aberdeen AB25 1HG
Email: rab,smc@scms.rgu.ac.uk
Tel: +44 1224 262702 Fax: +44 1224 262727
[2] ZENECA Pharmaceuticals, Hurdsfield Industrial Estate,
Macclesfield, Cheshire SK10 2NA
Tel: +44 1625 582828 Fax: +44 1625 501887

Abstract. The KRUST refinement tool has already been successfully applied to a variety of relatively simple classificatory problems, and a generic refinement framework is being developed. This paper describes the application of KRUST to a design system TFS, whose task is tablet formulation for a major pharmaceutical company. It shows how novel components can be included within KRUST's underlying knowledge model, and how KRUST's refinement mechanisms can be extended as required, by adding new operators to the existing toolsets. New mechanisms have been added whereby proofs of related examples are used to constrain and guide KRUST's refinement generation. TFS has provided valuable widening experience for attaining our eventual goal of developing a framework for a generic knowledge refinement toolkit.

Keywords:
Knowledge Refinement, Knowledge Maintenance, Design Application.

1 Introduction

Knowledge refinement is the process of correcting errors in the rulebase of a Knowledge-Based System (KBS), triggered when test cases are wrongly solved by the KBS. This paper describes how the knowledge refinement tool KRUST is being developed from a prototype, applicable to simple PROLOG rulebases, to a generic tool, applicable to industrial systems written in a variety of different shells. We demonstrate the feasibility of this approach by showing how KRUST has been applied to the Product Formulation Expert System (PFES), a shell which differs in a number of significant ways from the backward-chaining diagnostic shells which have typically been the target both of KRUST and many other refinement tools.

We first describe KRUST, and our current generic approach to the representation of knowledge and the use of refinement operators. We then introduce the expert system shell PFES, and a particular industrial application TFS, used on a regular basis by the pharmaceutical company Zeneca. We then show how the generic framework is able to handle two necessary extensions required by PFES.

First, we show how rule elements peculiar to PFES can be accommodated in KRUST's knowledge hierarchy. Secondly, we introduce new refinement operators and a new filter whereby traces of sets of related examples can be used to generate and filter refinements. These operators are the first KRUST procedures to employ induction; up to now, we have concentrated on the use of control information because of its importance for real expert system shells, but we believe induction is also essential.

We then demonstrate the effectiveness of KRUST's approach by presenting the results of applying KRUST to TFS, and showing that KRUST is able to fix actual bugs which occurred in an early version of the system. Finally, we compare KRUST with other tools, and conclude with plans for the next stage of our work on PFES.

2 Krust

The operation of any refinement system may be broken down into the following three tasks: *Blame allocation* determines which rules or parts of rules might be responsible for the erroneous behaviour; *Refinement generation* suggests rule modifications that may correct the erroneous behaviour, and *Refinement selection* picks the best of the possible refinements according to some criteria.

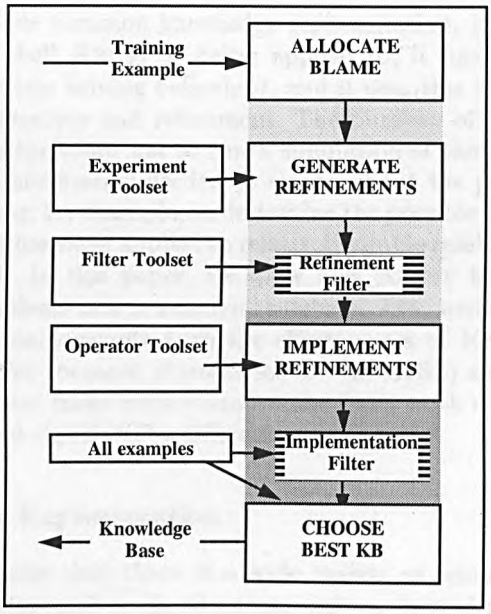

Fig. 1. The operation of KRUST

KRUST's input is a faulty KBS together with a set of examples, some of which are wrongly solved by the KBS. The refinement task consists of correcting the

KBS so that it correctly solves as many of the examples as possible. Figure 1 shows KRUST using one training example at a time to generate refinements, while the remaining examples are used to filter these refinements. More details of KRUST appear in [1].

2.1 The Development of a Generic Refinement Tool

A *generic refinement tool* [3] should satisfy three requirements:

- applicable to a variety of commercial shells;
- have a unified framework (i.e., not just a collection of separate refinement tools); and
- the framework should be extensible to apply to new shells.

The two principle features of the tool which will enable us to satisfy these requirements are:

- the ability to create an internal representation (or "knowledge skeleton") for each rulebase, using a common knowledge representation hierarchy for rule elements; and
- toolsets of filters and refinement operators, as shown in figure 1.

When discussing our common knowledge representation, it is important to note that, whatever shell KRUST is being applied to, it has direct access to the shell's actual problem solving behaviour, and it uses this information when performing blame allocation and refinement. The purpose of constructing the knowledge skeleton is therefore *not* to run a simulation of the shell, but rather to provide the extra information needed to reason about the possible effects of changes to the rule-base; for example, to determine the possible chaining of rules.

Up to now, KRUST has been applied to relatively simple rulebases in PROLOG, CLIPS, and KAPPA[9]. In this paper, we show how KRUST has recently been applied to fix real problems in a commercial rulebase, TFS, written in a different shell, PFES. We thus demonstrate both the effectiveness of KRUST's approach to knowledge refinement (because it works for a "real" KBS) and the feasibility of the generic refinement framework (because the PFES work required KRUST's extension to a new and significantly different type of shell).

2.2 Rule Element Representation

Although it first appears that there is a wide variety of representations used by various KBS development tools, there are only a limited number of roles that a rule element (condition or conclusion) plays within a rule. For example, a condition can succeed or fail, bind variables, or be involved in rule chaining. These roles are the basis of KRUST's hierarchy of rule element types.

[3] We have recently been awarded an EPSRC grant to extend our basic mechanism to develop such a generic refinement framework.

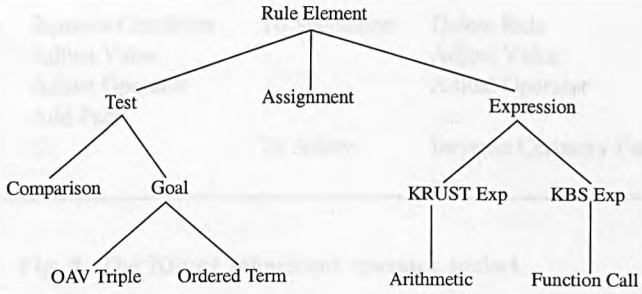

Fig. 2. KRUST's hierarchy of rule elements

A *test* is a rule condition that succeeds or fails. Tests are currently of two types: comparisons, such as inequalities, and goals. A rule element is a *goal* if and only if it can be involved in rule chaining. Two types of goal so far exist: *oav_triples* (terms of the form "Attribute of Object is Value") and *ordered_terms*, which consist of a keyword followed by arguments. Two ordered_terms unify if they have the same keyword and arity, and the corresponding arguments unify. *Assignment* has the obvious meaning. *Expressions* are rule elements that return a value. This class is divided into *KRUSTExps*, which can be evaluated within KRUST and *KBSExps*, which must be passed back to the KBS.

This hierarchy has grown during the course of the development of KRUST, and we expect that further terms will be added in the future as new KBS shells require. However, the use of a hierarchy allows us to add new rule elements within a consistent framework, and to implement procedures which take advantages of properties shared between different elements.

2.3 The Operator Toolset

To illustrate the purpose of the various toolsets shown in figure 1, we describe briefly the operator toolset, shown in more detail in figure 3. Here the labels on the left of the each column are *experiments*; that is, high-level descriptions of changes that might be made to the rule-base. Associated with each experiment are the names of the operators which actually implement these changes and create new, modified rule-bases. To add a new operator, an entry must be added in the table, and an associated Lisp function defined. For example, the new procedure described in §5.1 for learning facts from traces was implemented as an operator called *Add Fact* and associated with the experiment type *Generalise*.

3 PFES and TFS

3.1 PFES

PFES [13] is a shell whose purpose is to solve problems of design, synthesis or formulation, for which is it difficult to have access to all possible solutions in advance, so a solution must be synthesised or generated. Its control structure

To Generalise:	Remove Condition	To Specialise:	Delete Rule
	Adjust Value		Adjust Value
	Adjust Operator		Adjust Operator
	Add Fact		.....
	.....	To Allow:	Increase Certainty Factor
			.....

Fig. 3. The KRUST refinement operator toolset

is task-based, and corresponding to each task is a rule-set which is executed by forward-chaining. There is considerably less experience in building and refining such systems than for diagnostic ones. Since PFES can also be used to create traditional diagnostic systems, we believe that the application of KRUST to PFES represents a real extension to the power and applicability of refinement tools.

3.2 TFS

Drug formulation is a hard synthesis problem, and there are few formulation KBSs in regular commercial use [4]. One of these is the Tablet Formulation System (TFS) written in PFES[4], which solves the problem of selecting the inert substances, or *excipients*, which are needed to process a drug into a tablet [12]. The difficulty of the formulation task arises from the need to select a set of mutually compatible excipients, while at the same time satisfying a variety of other constraints.

The user provides a drug's name and its desired dosage. Then TFS calculates a *formulation* consisting of the most appropriate material from each excipient type, and the quantity of each required. During the initial stages of this process, TFS also calculates some intermediate results called the *specification*; these are necessary properties of the formulation which follow directly from the user's requirements. TFS input is thus drawn both from the user, and from databases containing chemical properties of drugs and excipients. TFS's output consists of the specification and formulation for the desired tablet. Figure 4 shows a typical example for KRUST, comprising requirements, specification and formulation.

3.3 How Refinement is Applied to TFS

There exist three versions of TFS: TFS-1B simply fixed a number of bugs in TFS-1A, but TFS-2, as well as fixing further bugs, represents a paradigm-shift in the approach to tablet-formulation. Our work on refining TFS is therefore divided into two phases. During the first phase, described in this paper, we applied KRUST to TFS-1A, using TFS-1B as an oracle to critique TFS-1A's output. KRUST has access to TFS-1A's rulebase, and traces of its behaviour on examples.

[4] We are grateful to Paul Bentley of Logica Cambridge Ltd. for his assistance with the PFES software interface.

TFS Input
Requirement: Drug: Drug-A, Dose: 60 mg., No of fillers: 2

TFS Output
Specification:
full-stability: Yes
drug-filler–concentration: 0.9
minimum-tablet-weight: 100mg
maximum-tablet-weight: 800mg
target-tablet-weight: 260mg
start-strategy: strategy-A
filler-concentration: 66.9%
typical-disintegrant: Maize-starch
disintegrant-concentration: 0.05
tablet-weight: 252.2mg
total-concentration: 97%
tablet-diameter: 8.73mm
... various other properties...

Formulation:
Tablet weight: 250mg
Fillers: Lactose 66.7%,
Calcium phosphate 2.4%
Binder: Gelatin 4.1%
Lubricant: Stearic acid 1.0%
Disintegrant: Croscarmelose 2.1%

Fig. 4. KRUST example, made up of TFS input and output

In contrast, the oracle provides just the correct output for each example input. In the next phase, we will regard TFS-1B as the buggy system to be refined, and TFS-2 as the oracle.

4 Applying KRUST to PFES

This has required additions both to the rule element hierarchy and to the sets of operators. However, most of the existing operators were found to be applicable to PFES, and the new operators required by PFES turned out to be applicable to the other shells, so that much of the work was not PFES-specific, thereby confirming our belief in a framework approach.

4.1 Rule Element Representation

Many of PFES's rule elements (conditions and conclusions) are standard expressions found in many KBS shells. However, there is a group of rule elements that appear at first unique to PFES, and therefore potentially difficult to represent within a common framework. These *agendas* are untyped lists, where items can be read and written to the top or bottom, or directly below another given item. Agendas are used to pass data between routines that generate values and those that subsequently test or filter them.

However, TFS agendas can also be interpreted as a mechanism for storing attribute-value data. Not all agendas have the same semantics, but the number of different possibilities actually employed within TFS is fairly limited. Two of the most common types are shown in figure 5. Each example shows the contents of an agenda at some point during the running of TFS-1A, together with the

PFES rule elements that write to and read from the agenda, and the KRUST representation of these elements.

The Filler-Agenda is simply a list of excipients; their presence on the agenda indicates that they have passed a stability test.

The Property-Agenda again shows a list of excipients, but now each excipient has an associated floating-point number, representing the value of a mechanical property.

In each of these cases, the rule elements which read and write the agenda items can be represented in KRUST as ordered terms. The fact that <STABILITY> is a property of <FILLER> is implicit in the PFES statements, which write a filler to the agenda, followed by its stability. However, it is made explicit in the KRUST representation, where the **agenda-unlabelled-attribute** statement includes both the filler and its stability as arguments. One consequence of this is that PFES commands of the type **add** <ITEM> **to-bottom-of** <AGENDA> have different KRUST representations, depending on whether or not <ITEM> represents an attribute. Fortunately it is possible to determine the correct translation from the context, both in the situations described here, and in other more complex situations also arising in TFS; this enables our translator to construct the correct KRUST representation automatically.

4.2 PFES's Many-Valued Output

One consequence of PFES's formulation task is that its output is a compound answer (figure 4), in contrast to the single result typically output from a diagnostic system. TFS-1A's output typically differs from the correct values at only one or two points, but some examples have 12 points of difference. Attempting to fix all of these at once leads to a combinatorial explosion, so instead KRUST automatically determines dependencies between faults, and attempts to fix the earliest fault(s) in the dependency chain first, in the hope that this will fix the later faults as well. A *dependency chain* is a sequence of rules where each conclusion matches a condition of the next rule, and a fault lies in the chain if it matches the conclusion of a rule in the chain. This technique has been implemented and applied to those few TFS-1A examples which exhibit large numbers of errors. In these cases, it was possible to obtain a few values that are prior to all the others in terms of dependency, and so KRUST focuses its repairs on them.

5 Learning from Traces

As stated above, KRUST has direct access to the actual rule execution behaviour of a KBS. For PROLOG, CLIPS and KAPPA rulebases, it queries the KBS directly. For PFES, this is not possible, so KRUST derives equivalent information from the *trace*. This is a record of every attempt by the rule interpreter to execute a rule element, together with the result and the bindings for all the variables. It thus

Snapshot of FILLER-AGENDA

CELLULOSE LACTOSE CALCIUM-PHOSPHATE

PFES Read/Write Operations on FILLER-AGENDA

Conclusion: **add** <FILLER> **to-bottom-of** FILLER-AGENDA
Condition: <FILLER> **is-on** FILLER-AGENDA

Krust representation of each operation

on-agenda(FILLER-AGENDA, <FILLER>)

Snapshot of PROPERTY-AGENDA

CELLULOSE 46.2 LACTOSE 158.8 CALCIUM-CARBONATE 851.1

PFES Read/Write Operations on PROPERTY AGENDA

Conclusions: **add** <FILLER> **to-bottom-of** PROPERTY-AGENDA
add <STABILITY> **to-bottom-of** PROPERTY-AGENDA
Conditions: <FILLER> **is-on** PROPERTY-AGENDA
<STABILITY> **is-the-item-after** <FILLER> **on** PROPERTY-AGENDA

Krust representation of each of these paired operations

on-agenda(PROPERTY-AGENDA, <FILLER>)
agenda-unlabelled-attribute(PROPERTY-AGENDA, <FILLER>, <STABILITY>)

Fig. 5. Agendas and their PFES operations

provides the information needed to construct a proof-tree for the conclusions generated, which is then used for blame allocation and refinement generation.

Earlier versions of KRUST have generated refinements for a single training example at a time, subsequently using other examples for filtering and judging. We describe two recent additions to KRUST which enable it to use *sets* of traces from *related* examples to guide the refinement process at an earlier stage. The approach in both cases is as follows:

- to select as positive examples those examples and their traces that exhibit a particular fault, and as negative examples those that do not;
- to identify features distinguishing these two groups; and finally
- to use these features as inputs to refinement generation and selection.

We shall describe two techniques: using traces to learn facts, and for refinement filtering. The goal of the second technique is to reduce KRUST's search space, so we shall postpone its description until §7, after the results *without* the new filter. The remainder of this section is therefore devoted to the first technique: using traces to learn facts.

5.1 Using Traces to Learn Facts

Most of KRUST's refinements take the form of modifications to existing rule elements. However, one of the available refinement operators adds a new fact to the database. Such a refinement is generated when the new fact is required to satisfy a currently unsatisfied condition in a rule, and hence enable that rule to fire. Induction from several examples should be applied at this point, because typically the training example imposes insufficient constraints on the new fact.

We now illustrate this technique with reference to faults of type 1, in which TFS-1A incorrectly recommends the filler Calcium Phosphate. The reason is that the MAX-LEVEL of Calcium Phosphate is missing from the data-base, so that the second rule, **Remove-Excessive-Fillers**, fails to fire and hence fails to remove Calcium Phosphate from the filler agenda (figure 6). Hence Calcium Phosphate remains on the agenda to be read by **Get-Insoluble-Filler**.

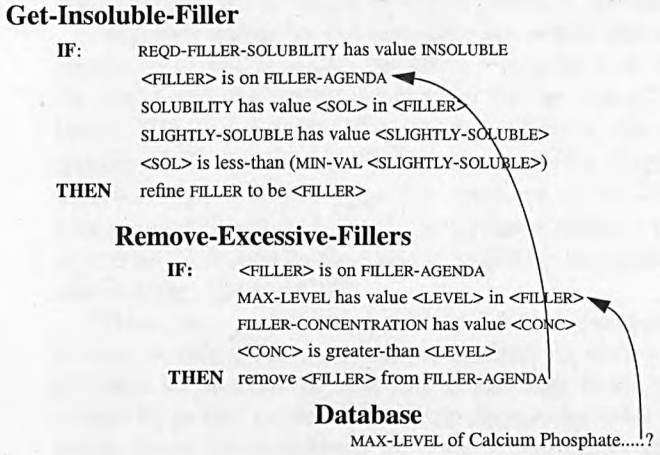

Fig. 6. Rule chain for wrong filler example

KRUST identifies the missing database value as one possible cause of the error by backward chaining from the erroneous conclusion: <FILLER> = **Calcium Phosphate**. One way of *preventing* the rule **Get-Insoluble-Filler** from firing for Calcium Phosphate is to *enable* the rule **Remove-Excessive-Fillers** to fire. To do so, it is necessary to enable the condition: MAX-LEVEL **has value** <LEVEL> **in** <FILLER> which can only be done by adding an entry for MAX-LEVEL of Calcium Phosphate to the database. KRUST has three sources of constraints on the value to which MAX-LEVEL should be set:

1. The current training example.
2. Other positive examples of the same fault — that is, examples for which TFS-1A wrongly recommends Calcium Phosphate.
3. Negative examples of the fault — i.e., examples in which TFS-1A correctly recommends Calcium Phosphate.

We consider first the way in which the current training example and the other positive examples are used. In the following discussion, rule R_f is the rule whose failure is being corrected, so here, R_f is **Remove-Excessive-Fillers**. For the positive examples, KRUST first checks that rule R_f failed on the same condition as for the training example, and thereafter processes the training example and the other positive examples identically. KRUST must first determine the constraints on MAX-LEVEL that must be satisfied for R_f to succeed. Here, the value of MAX-LEVEL is stored in the variable <LEVEL>, and there is only one constraint: <CONC> **is greater-than** <LEVEL>. The value of <CONC> is not available from the trace of R_f, as the rule interpreter never reached this condition; however, the rule that set the value of <CONC> will have fired, and the value of <CONC>, and hence the constraint imposed on MAX-LEVEL may be obtained from its trace. Thus the training example and the other positive examples provides a set of bounds on MAX-LEVEL. Here, these are all upper bounds, which reduce to the single constraint **58% is greater-than** MAX-LEVEL.

Negative examples are examples for which the system gives the same conclusion as in the training case (here, <FILLER> = **Calcium Phosphate**) but for which the conclusion is accepted by the expert, and for which the rule R_f failed. KRUST constrains the new fact added to the database so that it will not interfere with the proofs of these examples by causing R_f to succeed. It must therefore ensure that at least one condition of R_f will continue to *fail*. Here, the relevant condition is <CONC> **is greater-than** <LEVEL>, so that the values of <CONC> arising in the various negative examples provide a *lower* bound for MAX-LEVEL, (here, 11.5%).

These two constraints define an interval delimiting the new value, and can be used in one of two ways: they can either be shown to an expert, who can then be asked to provide the new fact or else adjudicate on any inconsistencies (e.g., caused by noise); or else KRUST can choose the value from within the constraints whose insertion constitutes the most conservative change. In our example, this will be the greatest value for MAX-LEVEL of Calcium Phosphate that lies within the bounds induced from the examples; i.e., 57%.

6 Results with TFS-1A and TFS-1B

Since our work is designed to solve problems of knowledge-base refinement rather than machine learning [2], the traditional machine learning approach involving learning graphs is inappropriate. KRUST, like MOBAL [6], is not well described by the paradigm of "learn a concept and then use this concept to classify previously unseen examples". Rather, its purpose is to refine real KBSs, with real faults, and to generate real repairs. TFS-1A was not intended to be faulty (i.e., we did not introduce artificial faults for KRUST to find), and TFS-1B contains the actual fixes for those bugs. The goal of our experiments was thus to determine whether KRUST was able to fix these faults in TFS-1A.

We first generated 208 TFS inputs, evenly distributed over the requirement space, and ran TFS-1A and TFS-1B on each of them. Any difference between

the outputs of the two system for a given input constituted a fault in TFS-1A. A hierarchical clustering mechanism grouped these faults into the classes shown in figure 7. These in fact correspond to the three differences between TFS-1A and TFS-1B. We now describe a set of typical runs where we have selected one example for each of the faults.

Fault 1: Incorrect filler

Example 5 — (110 mg, DRUG-A, 1 filler)
1A *filler:* Calcium phosphate 57.3% 1B *filler:* Calcium carbonate 58.5%

Example 33 — (110 mg, DRUG-F, 1 filler)
1A *filler:* Calcium phosphate 55.5% 1B *filler:* Magnesium carbonate 56.6%

Fault 2: Incorrect quantity of binder

Example 1 — (10 mg, DRUG-A, 1 filler)

1A *binder:* Gelatin 4.1%	1B *binder:* Gelatin 2.1%

Fault 3: Multiple faults in specification produced

Example 183 — (360 mg, DRUG-A, 1 filler)

1A *target tablet weight:* 400mg	1B *target tablet weight:* 450mg
1A *drug concentration:* 9/10	1B *drug concentration:* 4/5
1A *filler concentration:* 0.0	1B *filler concentration:* 0.1
... *various other discrepancies*	

Fig. 7. Contradictory formulations generated by TFS-1A and TFS-1B

6.1 Fault 1: Wrong Filler

One of the 61 refinements involved the induction module (§5.1). It returned a value of 57%, but this was overridden when the expert provided a value of 30%.

All but one of the refined knowledge bases (KBs) were rejected at the first filtering stage because they did not give the correct output for the training example. The refined KB recommended by KRUST coincided with the "correct" TFS-1B KB, and was the original TFS-1A KB, with the addition of the following database entry: MAX-LEVEL **of Calcium Phosphate is 30%.**

6.2 Fault 2: Wrong Binder Level

93 refinements were generated. Again, only one of the refined KBs passed the first filtering stage. The KB recommended by KRUST coincided with the "correct" TFS-1B KB and was the original KB, except that the conclusion to the rule **Default-Binder-Level** had been changed from **Set the value of** <BINDER>

in the FORMULATION **to be 0.04** to **Set the value of** <BINDER> **in the** FORMULATION **to be 0.02**. Again, this proved to be the correct fix, the refined KB coinciding with the TFS-1B KB.

The effectiveness of evaluating the refined KBs on the training example for faults 1 and 2 at first seems surprising, since for other applications it is common for many refined KBs to pass this stage, necessitating further filtering. However, TFS's output is a complex formulation, so that an incorrect perturbation of TFS-1A's KB is far less likely to lead to a correct solution for the original training example. This is one benefit of refining a design system, and contrasts with our experience with diagnostic systems, where the difficulty is to select appropriate test examples with which to filter the many refined KBs that are generated [10].

6.3 Fault 3: Multiple Faults in Specification

Since the system and oracle outputs differed on multiple fields for this fault, the technique described in section 4.2 was applied to these fields, and selected **Target-Tablet-Weight** as the independent one. It turned out that all the cases of multiple faults in the specification were in fact caused by an error in an equation in the rule **1st-Guess-Weight**. KRUST was unable to fix this error, because it currently has no operators for transforming equations.

RULE **1st-Guess-Weight**
IF DRUG has value <DRUG> in the FORMULATION
AND <DRUG> has value <DOSE> in the FORMULATION
AND <WEIGHT> = (ROUND-TO-NEAREST 5 <DOSE> / 0.1 + (0.00221 * <DOSE>))
THEN
Set the value of TARGET-TABLET-WEIGHT in the SPECIFICATION to be <WEIGHT>

KRUST was nonetheless able to *identify* the faulty rule, and generate 3 refinements to it, specialising each of the 3 conditions. The small number of refinements is explained by the fact that a single rule was responsible for the faulty conclusion, and no chaining was involved. KRUST also attempted to propagate the desired value of <WEIGHT> back through the equation in the third condition to give a corresponding value for <DOSE>, but this did not lead to a refinement, since the value of <DOSE> may not be altered.

The correct fix may be found in TFS-1B, where an additional rule uses a cubic equation to calculate TARGET-TABLET-WEIGHT for high values of <DOSE>. We believe functionally equivalent rules could be learned by an extension of the inductive techniques described in §5.1; this is work in progress.

7 Using Traces for Refinement Filtering

We now return to the use of traces to guide KRUST's behaviour, and introduce the second technique for the use of traces.

The refinements generated by KRUST are derived from the system's incorrect conclusion and its proof, together with the expert's conclusion. Therefore, if we

also consider proofs that are similar to the faulty proof but yet lack the fault, we may obtain new information useful to the refinement process. We have used this new information to filter out refinements that are unlikely to fix the fault. Our trace comparator takes a pair of traces, and compares the firing behaviour for each of the rules for which KRUST proposes a repair. The procedure for filtering is as follows.

1. Let R be the set of rules which KRUST is refining.
2. Select sets of examples F and C, where examples F exhibit the fault (that exhibited by the current training example) and C do not.
3. Run the comparator for each rule $r_k \in R$ and for each pair of examples $(f_i, c_j) \in F \times C$. We define the comparator function $\text{diff}(r_k, f_i, c_j)$ to be 1 if the firing behaviour for rule r_k differs for examples f_i and c_j, 0 otherwise.
4. Then we say that the behaviour of rule r_k is relevant to the fault iff $\exists\ j$ such that $\forall i\ \text{diff}(r_k, f_i, c_j) = 1$.

Note that the appearance of a fault in one example and not in another may arise in two ways:

- it could be that a certain rule r fires in one case and not in the other, or
- it could be that r's firing behaviour is the same in both cases, but that this behaviour is faulty in the first case but correct in the second case.

The comparator will detect the difference in the first case, but not in the second. Hence we cannot choose as a criterion of relevance that the behaviour of r should differ for *all* faulty/non-faulty pairs.

7.1 Results of Refinement Filtering

The trace comparison technique was applied after KRUST had already been run without it, as described in §6, so that it was already known which of the refinements generated was the correct one. The purpose of this experiment was to determine whether the technique could be used to remove irrelevant refinements at the refinement filtering stage (figure 1), thus avoiding the necessity for KRUST to write out the refined KBs, load them into PFES, and test them.

The algorithm was applied to the runs described in §6.1 and §6.2[5]. The second column in tables 1 and 2 indicates which rules were involved in the chaining process that lead to the faulty conclusion, and the third column shows which rules the trace comparator identified as potentially relevant to the fault. The tables show that, for both faults, the technique could be used to filter out some refinements that were indeed unrelated to the faulty conclusion, while not rejecting any that *were* relevant. The apparently poor behaviour for fault 1, where the technique highlighted only one of a possible three irrelevant rules, may be explained as follows. Fault 1 is a rarely-occurring fault, and all the

[5] For fault three, KRUST generated refinements to a single rule only, so the algorithm was not required.

examples of this fault happen to share certain other attributes: viz., they use no surfactant, and the drugs involved are soluble. These attributes are reflected in the firing behaviour of the rules related to these attributes (**Insoluble-Drug-Rule** and **Initial-Surfactant-Level**), so that the comparison algorithm also identified these rules as potentially relevant.

Rule	Involved in faulty conclusion	Trace comparator indicates relevance
Get-Soluble-Filler		
Insoluble-Drug-Rule		√
Get-Insoluble-Filler	√	√
Remove-Excessive-Fillers	√	√
Initial-Surfactant-Level		√

Table 1. Trace Comparator applied to Fault 1: Wrong Filler

Rule	Involved in faulty conclusion	Trace comparator indicates relevance
Default-Binder-Level	√	√
Update-Formulation	√	√
High-Dose-Binder-Level		√
Try-Dose-Again		
Find-Stable-Surfactant		
Default-Surfactant		

Table 2. Trace Comparator applied to Fault 2: Wrong binder level

7.2 Further Developments

These examples have illustrated how groups of traces from related examples can be used to reduce KRUST's search. However, we believe that these techniques will acquire even greater importance when we refine TFS-1B up to TFS-2, since this task will be substantially harder that the refinement of TFS-1A to TFS-1B. The reason for this is the nature of the "paradigm shift" exemplified by TFS-2. Its principle feature is the introduction of three categories to which all excipients are assigned. Formulators now have the further constraint of being required to choose excipients from the lowest possible category. Consequently the task of refining TFS-1B so that it behaves like TFS-2 will include the problem of learning rules which implement the new formulation policy. This is harder than any of the refinements required by TFS-1A, and will demand the use of induction. However, the extension of KRUST to extract as much information as possible from multiple related examples is well-suited to this type of problem.

8 A Comparison with Related Work

The program CLIPS-R [7] refines forward-chaining production systems written in CLIPS, and so has to deal with some of the problems discussed in this paper. CLIPS-R's approach to refinement is similar to KRUST's in that it identifies faulty rule elements by working backwards from observed faulty behaviour, but it also differs from KRUST in a number of ways.

1. It requires traces containing more information than PFES traces, such as the fact-list prior to each rule firing, and information linking facts and their sources.
2. It has been applied to diagnostic but not design systems. It is not clear how well CLIPS-R's grouping of traces sharing an initial sequence of rule firings would apply to production rules functioning in the generate and test mode found in design problems.
3. It can use, though does not require, a variety of user-supplied constraints on the correct behaviour of the KBS.

Many theory revision tools such as EITHER [8] and FORTE [11] are restricted to PROLOG programs, and SEEK [5], TEIRESIAS [3] and ODYSSEUS [14] are each applicable only to a single shell. KRUST on the other hand is currently usable with PROLOG, CLIPS, KAPPA and PFES applications. Other tools select refinements before implementing them as new KBs, so may not detect unintended side-effects. KRUST is unusual in that it generates and tests many refined knowledge bases.

Some tools can reason about a shell's control strategy (for example, ODYSSEUS makes use of meta-rules to guide the learning of new object rules) but they are often tied to a particular shell. KRUST can deal with backward and forward-chaining rules, can reason about a rule's priority under conflict resolution, and is not tied to one particular shell. This ability is due both to the explicit representation of control information, and to the fact that KRUST has direct access to the shell's actual problem solving behaviour. This contrasts with EITHER's approach, which is to treat a PROLOG program as a logical theory, ignoring the ordering of rules, and hence also the order in which solutions are generated. This approach is less suited than KRUST's to the refinement of real systems, where conflict resolution and rule ordering are usually important aspects.

Knowledge refinement tools such as KRUST also require fewer examples, although theory revision tools are more adept at adding new knowledge, based on many examples. CLIPS-R shares properties of both types of tool, and our current work on KRUST is aimed at giving it some of the desirable properties of theory revision tools, such as induction, while continuing to develop its ability to cope with real-world systems.

9 Conclusions

The experience of applying KRUST to a PFES KBS indicates that its basic refinement techniques are equally applicable to design systems. It also confirms our

belief that a relatively small set of basic knowledge components are commonly used in KBSs, and also that novel ones may be fairly closely related to existing ones; i.e., the roles of knowledge components (not the knowledge itself) are fairly limited. This also suggests that our goal of a more general refinement framework is feasible. One gain of refining a design system is the relative complexity of the conclusion, which proves very helpful in isolating relevant rules and hence repairs. We also take advantage of the knowledge contained in PFES traces. Information from multiple traces guides and constrains refinement generation. Traces also allow KRUST to propose new knowledge, a feature more usually associated with theory revision systems. Both these techniques will be increasingly relevant when KRUST must learn the more complex concepts required in the most recent version, TFS-2.

References

1. Susan Craw and Paul Hutton. Protein folding: Symbolic refinement competes with neural networks. In Armand Prieditis and Stuart Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 133–141, Tahoe City, CA, 1995. Morgan Kaufmann.
2. Susan Craw, Derek Sleeman, Robin A. Boswell, and Leonardo Carbonara. Is knowledge refinement different from theory revision? In Stefan Wrobel, editor, *Proceedings of the MLNet Familiarization Workshop on Theory Revision and Restructuring in Machine Learning (ECML-94)*, pages 32–34, Catania, ITALY, 1994. GMD Technical Report Number 842.
3. R. Davis and D.B. Lenat. *Knowledge-Based Systems in Artificial Intelligence.* McGraw-Hill, 1982.
4. Jürgen Frank, Birgit Rupprecht, and Weit Schmelmer. Knowledge-based assistance for the development of drugs. *IEEE Expert*, 12(1):40–48, 1997.
5. Allen Ginsberg, Sholom M. Weiss, and Peter G. Politakis. Seek2: A generalized approach to automatic knowledge base refinement. In *Proceedings of the Ninth IJCAI Conference*, pages 367–374, 1985.
6. Katharina Morik, Stephan Wrobel, Jörg-Uwe Kietz, and Werner Emde. *Knowledge Acquisition and Machine Learning.* Academic Press, London, 1993.
7. Patrick M. Murphy and Michael J. Pazzani. Revision of production system rule-bases. In W. W. Cohen and H. Hirsh, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 199–207, New Brunswick, NJ, 1994. Morgan Kaufmann.
8. Dick Ourston and Raymond Mooney. Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309, 1994.
9. Gareth Palmer. Applying KRUST to a new KBS tool: experience with Kappa. Technical Report 95/9, SCMS, Robert Gordon University, October 1995.
10. Gareth J. Palmer and Susan Craw. The selection of training cases for automated knowledge refinement. In *EUROVAV-97*, 1997.
11. Bradley L. Richards and Raymond J. Mooney. Refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.
12. Ray Rowe. An expert system for the formulation of pharmaceutical tablets. *Manufacturing Intelligence*, 14:13–15, 1993.

13. John Turner. Product formulation expert system. *Manufacturing Intelligence*, (8):12 – 14, 1991.
14. David C. Wilkins. Knowledge base refinement as improving an incorrect and incomplete domain theory. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning Volume III*, pages 493–513. Morgan Kaufmann, San Mateo, CA, 1990.

This article was processed using the LaTeX macro package with LLNCS style

Appears in *Proceedings of the 9TH IEEE International Conference on Tools with Artificial Intelligence*, pp446–453, IEEE Press, 1997

Knowledge Refinement to Debug and Maintain a Tablet Formulation System

Susan Craw Robin Boswell*
School of Computer and Mathematical Sciences
The Robert Gordon University
Aberdeen AB25 1HG, UK
Email: smc,rab@scms.rgu.ac.uk

Ray Rowe
ZENECA Pharmaceuticals
Hurdsfield Industrial Estate
Macclesfield
Cheshire SK10 2NA, UK

Abstract

Knowledge refinement tools have commonly been applied to diagnostic applications. This paper considers the refinement of a design application. It explores the differences in knowledge content and problem-solving steps for design rather than diagnosis systems, and investigates additional refinement operators. Although the problem-solving task itself tends to be more complex, this in fact puts more constraints on the results of the problem-solving and so the evidence on which the refinement is based can be more rich. Results are provided for a real tablet formulation system, TFS*, and experience is reported for two types of refinement tasks: debugging to correct faulty problem-solving of an early version of* TFS*; maintenance of* TFS *when the formulation task is altered by a change in company policy.*

1. Introduction

Knowledge refinement is the process of changing knowledge in a knowledge-based system (KBS) in reaction to evidence that the KBS is not producing correct solutions. The KBS is evaluated on tasks provided by an expert. Those where the solutions of the KBS and expert are incompatible provide evidence that the KBS is faulty. The "correct" task-solution pairs can be regarded as training cases for the learning undertaken within the refinement process.

Automated knowledge refinement comprises the following steps. **Blame Allocation** identifies rules or individual conditions which are responsible for the wrong solution. These may be rules which have fired and so are part of the explanation for the solution: **error-causing rules** have fired when they should not have, **wrong-fire** rules have fired but provided the wrong value. However, blame allocation must also identify rules that have not fired but, if they were used, would produce the desired expert solution (**target rules**). **Repair Generation** decides which knowledge changes effect repairs that prevent error-causing rules from firing, allow wrong-fire rules to fire correctly or enable target rules to fire. Of course there will be many changes that achieve similar effects. **Refinement Selection** chooses which potential repairs are likely to be the most effective overall. Selection happens at various stages of the refinement process: which rules are most likely to be to blame for the faulty training case, which repair is most likely to achieve improved problem-solving, which refined knowledge base (KB) appears to achieve the best problem-solving? Selection is therefore based on heuristic knowledge about effective blame allocation or refinement generation, or empirical evidence of the effective problem-solving of refined KBs.

Our application is a rule-based Tablet Formulation System, TFS, implemented in Logica's Product Formulation Expert System shell PFES [16]. TFS is in routine use at Zeneca Pharmaceuticals, where it forms an important stage in the development of tablet formulations for new drugs [15]. Tablet formulation is a complex synthesis task and TFS is one of the few knowledge-based formulation systems in regular commercial use [6]. We are particularly interested in working with TFS because there is considerably less experience both in building and refining such design systems than for the more traditional diagnostic systems.

Section 2 introduces TFS. The KRUST refinement tool is described in Section 3, with TFS-specific details appearing in Section 4. Our experience applying KRUST to TFS is described in Section 5. Related work is considered in Section 6 and Section 7 contains conclusions.

2. The tablet formulation application

The design of a new tablet involves identifying other substances that must be included in the tablet's recipe so that the tablet is manufactured in a robust form, and the desired dosage of drug is delivered and absorbed by the patient. This involves the identification and quantities of in-

*This work was supported by a Robert Gordon University studentship and in part by EPSRC grant GR/L38387

ert substances called **excipients** to balance the properties of the drug [15]. Excipients play the role of fillers, binders, lubricants and disintegrants in the tablet. The difficulty of the formulation task arises from the need to select a set of mutually compatible excipients, whilst at the same time satisfying a variety of other constraints.

The input to TFS is a **requirement** comprising the drug name and desired dose, together with the number of fillers to be used. TFS produces a **specification** indicating the desired properties of the tablet and the **formulation** for the tablet:

TFS Requirement

DRUG: Drug-A DOSE: 60 mg
NUMBER OF FILLERS: 2

TFS Specification

FULL-STABILITY: Yes
TARGET-TABLET-WEIGHT: 260mg
START-STRATEGY: Strategy-A
FILLER-CONCENTRATION: 66.9%
DISINTEGRANT-CONCENTRATION: 0.05
TABLET-DIAMETER: 8.73mm
... various other properties ...

TFS Formulation

TABLET WEIGHT: 250mg
FILLERS: Lactose 66.7%
Calcium Phosphate 2.4%
BINDER: Gelatin 4.1%
LUBRICANT: Stearic acid 1.0%
DISINTEGRANT: Croscarmelose 2.1%

We have access to several versions of TFS. **TFS-1A** is an initial version, corresponding to an early stage in development. Its KB is buggy and produced faulty tablet formulations for some tablet requirements. **TFS-1B** is a manually debugged version of TFS-1A. TFS-1B produced correct formulations during its period of usage. **TFS-2** is a manually updated version of TFS-1B that resulted from a paradigm shift in the company's approach to tablet formulation. It also contains further bug repairs. TFS-2 produced correct formulations according to the revised formulation practice during its period of usage. It should be noted that TFS-2 was an update rather than a completely re-engineered system. This evolution of TFS demonstrates the need for debugging and maintenance in the development of a KBS in a commercial environment. Therefore knowledge refinement has a role to play in both the debugging of faulty knowledge and the updating of knowledge to reflect changes in the operational environment.

3. The KRUST tool

Our KRUST refinement tool is unusual in generating many refined KBs and postponing the final choice of a recommended refined KB until it undertakes an empirical evaluation. Figure 1 shows a typical application of KRUST. A set of training cases is available each of which consists of a task and the expert's solution. For TFS, each training case consists of a requirement, a specification and a formulation as described in Section 2. Individual wrongly solved training cases are used to trigger the refinement cycle, but the set of training cases is used to induce new knowledge, reject unsuitable refined KBs, and evaluate the suitable refined KBs in order to select the preferred one, possibly for subsequent refinement with later training cases.

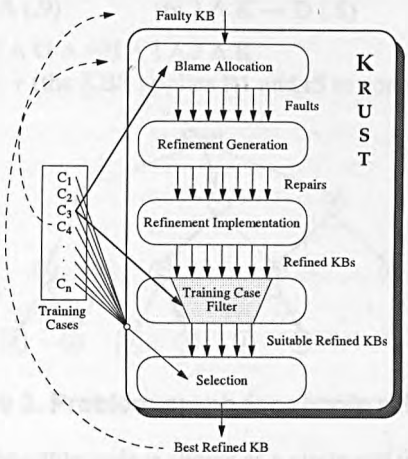

Figure 1. A typical application of KRUST

KRUST is particularly applicable to KBSs where inference provides additional control over the exhaustive deduction assumed in logical theories. It takes account of a rule's position on the execution agenda as well as considering whether the rule's conditions are satisfied or not. Therefore, as well as proposing changes that affect the logical content of the rules, it also suggests knowledge changes that affect how the rule is handled by the control mechanism.

KRUST creates internal models of the KBS's knowledge and its problem-solving (Figure 2). This simulation of the KBS is not used as an efficient replacement of the KBS [3]. Instead it is an aid to guide the refinement process; the refinements are checked using the actual KBS. In this way KRUST reasons about the refinement process in a common knowledge format but does not rely on duplicating all nuances of the KBS.

3.1. Modelling knowledge: the knowledge skeleton

The **knowledge skeleton** is an internal frame-based representation of the KBS's knowledge. It is a general representation of the knowledge, but can be fairly simple since it must faithfully represent only those portions of the knowledge that are pertinent to refinement with KRUST; a language like Ontolingua is unnecessary. KBS validation work

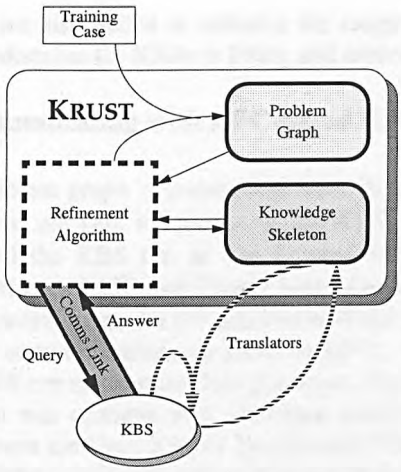

Figure 2. KRUST and KBS processes

also adopts this view [13]. Knowledge that should not be refined (such as procedural code to interact with the user) is duplicated but marked as irrelevant for refinement. A grammar for the relevant knowledge types in the shell's rule format generates a translator that creates a knowledge skeleton for the KB, and reconstructs a refined KB from the original KB and changes in the skeleton, Figure 2.

The knowledge skeleton represents rules as combinations of rule elements. It allows navigation between related pieces of knowledge (rule chains) and assembles additional refinement specific knowledge: which refinement operators are applicable, links to repairs for the rule element. We have evolved an expandable hierarchy of rule elements where a small set of basic rule elements are identified with the various knowledge formats used in several shells. In practice, a rule element conforms to one of a very small set of roles: logical n-ary predicate, variable assignment, numerical test, arithmetic calculation, procedure call.

3.2. Modelling inference: the problem graph

The problem-solving of a KBS is affected by both the knowledge content and the inference engine. Therefore, we have always believed that it is important to consider refinements that alter the way the knowledge is handled by the inference engine, in addition to the more standard changes to the content of the knowledge. Our work with Prolog KBs has always assumed that "the conclusion" is the first solution found and hence has taken account of Prolog's rule order conflict resolution and depth first search [4]. Therefore some refinement operators are concerned with altering the position of rules in the KB. We have extended this idea by developing refinement operators for several shells; specifically CLIPS, PFES and PowerModel (formerly KAPPA).

The **problem graph** is a graphical representation of the problem-solving for a particular training case, that captures the content of the deductions and the structure of the rule firing agenda. Its nodes contain [rule element, deducibility] pairs and the links represent problem-solving. Figure 3 shows the problem graph for the following task-solution pair and simple backward chaining rules when high rule priorities win:

R1: C → – (.8) r2: G → A (.8)
R2: D → – (.7) r3: G ∧ H → B (.7)
R3: A ∧ B → + (.9) r4: H → C (.8)
R4: D ∧ E → + (.6) r5: I → C (.7)
r1: F → A (.9) r6: J ∧ K → D (.8)

Task: ¬F ∧ G ∧ ¬H ∧ I ∧ J ∧ K
Solution: + (the KBS applies R1 and r5 to conclude –.)

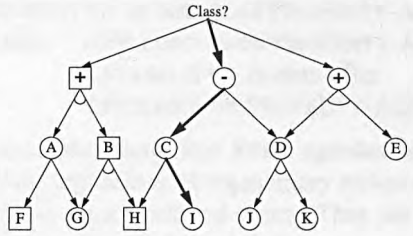

Figure 3. Problem graph for simple ruleset

Each deducible node is shown as a circle and unprovable ones are squares. A node's links are ordered to reflect the decreasing order imposed by conflict resolution and so the solution graph (bold) is the leftmost subgraph whose nodes are deducible. Potential KBS faults are identified in the problem graph as incorrect deductions or wrongly configured links. Repairs express changes to the problem graph as alterations to the knowledge skeleton. Thus, the knowledge skeleton and problem graph indicate the structure and processing of the KBS from which refinements are proposed. We believe this approach can be extended to more complex control mechanisms since Menzies [8] uses a similar abductive model for several problem solving methods.

It may appear that we are trying to create a generic **simulation** for a range of KBSs. This is not the case; the knowledge skeleton and problem graph are simply used to propose possible refinements. KRUST always generates refinements and empirically filters and evaluates them by communicating with the **actual** KBS. Therefore, mismatches between the actual KBS and the internal models result only in proposed refinements that fail and are removed during the empirical filtering, or missed refinements where one of the many other refinements will be selected instead.

4. Using KRUST with TFS

In this section we indicate the updates to KRUST that enable it to model, and so refine, PFES KBSs [2]. Refinement

operators are also added to enhance the range of repairs KRUST undertakes for KBSs in PFES, and other shells.

4.1. Communicating with a PC-based KBS

The problem graph is constructed from the knowledge skeleton and answers to queries to the KBS. Originally KRUST and the KBS ran as one process with communication between LISP and Prolog being handled by the POPLOG environment. CLIPS and PowerModel have APIs that allow communication via pipes or RPCs, so KRUST and the KBS run as separate Unix processes, Figure 2. This mechanism was changed to a file-based transfer of messages between the Unix KRUST process and PFES running on a PC. PFES's GUI was replaced by functions that communicated with the message-passing mechanism.

This simple communication was sufficient because few messages were needed since KBS behaviour was extracted from PFES traces. These contain a record of every attempt by the rule interpreter to execute a rule element, together with the result (success or failure) and the bindings for all its variables. The training case trace provides information to construct the problem graph, which is then used for blame allocation and refinement generation.

4.2. Designs as evidence of faults

TFS's formulation task produces a compound answer (Section 2), in contrast to the single result typical of diagnostic systems. Normally, the outputs from different versions of TFS differ at only one or two points, but some training cases can have as many as 12 points of difference. Combining the repairs to fix many differences will lead to a combinatorial explosion. Instead, KRUST automatically determines a chain of dependencies between faults, and attempts to fix the earliest first. A dependency chain is an ordering of the faults from the sequence of rule firings. For those few TFS training cases which exhibit large numbers of errors, it was possible to obtain a few values that are prior to all the others in terms of dependency, and so KRUST focuses its repairs on them.

4.3. New rule elements

Many of the rule elements in PFES KBSs are immediately identified with standard rule elements already used in KRUST's knowledge skeleton and so are easily expressed in the grammar that is used to generate the translators. But PFES rules also apply **agendas**. These are not standard rule execution agendas (e.g. as found in CLIPS), but are used to pass data between rules that generate values and those that subsequently test or filter them. This action is common in constructive KBSs where partial designs must be extended.

Agendas are untyped lists, where items can be read and written to the top or bottom, or directly below another item. Examples of the two most common types in TFS follow.

The **Filler-Agenda** is a list of excipients; their presence on the agenda shows they have passed a stability test:

Snapshot: Cellulose Lactose
PFES Conc:
add <FILLER> to-bottom-of FILLER-AGENDA
PFES Cond: <FILLER> is-on FILLER-AGENDA

The **Property-Agenda** contains a list of pairs comprising an excipient and the value of some mechanical property:

Snapshot: Cellulose 46.2 Lactose 158.8
PFES Conc:
add <FILLER> to-bottom-of PROPERTY-AGENDA
add <STABILITY> to-bottom-of PROPERTY-AGENDA
PFES Conds: <FILLER> is-on PROPERTY-AGENDA
<STABILITY> is-item-after
<FILLER> on PROPERTY-AGENDA

These examples show that PFES agendas behave like attribute-value tuples in working memory and so are similar in behaviour to CLIPS ordered terms. They are translated to KRUST's rule element for ordered terms:

Filler: on-agenda(filler-agenda, <Filler>)
Property: on-agenda(property-agenda, <Filler>)
agenda-anon-attribute(property-agenda, <Filler>, <Value>)

Notice that conclusions that add items to agendas have different representations in the skeleton, depending on whether the item is an attribute or value. The translator chooses the construction automatically from the context.

We have identified all PFES's rule elements with existing rule elements for KRUST's knowledge skeleton. However, if a KBS rule element does not correspond to existing rule elements, then a new type can be added to KRUST's rule element hierarchy, but new refinement operators must also be defined, based on its parent in the hierarchy.

4.4. New refinement operators

Earlier versions of KRUST concentrated on conservative changes to the knowledge, based on the evidence of a single training case, rather than creating new knowledge. This decision resulted from our desire to investigate how effective knowledge refinement is when few training cases are available; a common scenario with real applications. Therefore, although a refinement operator to create new rules existed, it simply built a rule whose conditions were the training case facts and whose conclusion was the expert's solution. Needless to say the addition of this (very specific) rule was rarely chosen as best in the evaluation of refined KBs!

We have now started to add inductive refinement op-

erators to learn novel pieces of knowledge from a set of training cases. Our experience with TFS has shown that although KRUST has always been able to **identify** these faults, induction is necessary to actually suggest repairs to the knowledge. Our domain expert was very content with KRUST identifying the faults and was happy to volunteer new knowledge for KRUST to incorporate. However we regarded TFS's need for inductive operators as a timely impetus to enhance KRUST's operators!

4.4.1. Acquiring missing database entries

In addition to IF/THEN rules, TFS contains a database of facts about drugs and their interaction with excipients. KRUST treats database entries as rules whose antecedent is always true and suggests refinements in the same way as for rules. New inductive operators are used to add missing facts to the database, but could also be used to create additional conditions for rules.

As an example, suppose TFS wrongly suggests Calcium Phosphate as a filler and KRUST's problem graph identifies the rule chain in Figure 4 as a fault. **Get-Insoluble-Filler** has fired because Calcium Phosphate is (still) on the filler agenda, since **Remove-Excessive-Fillers** has not removed it because the MAX-LEVEL of Calcium Phosphate is missing from the Database. One of KRUST's possible repairs is to supply the database value.

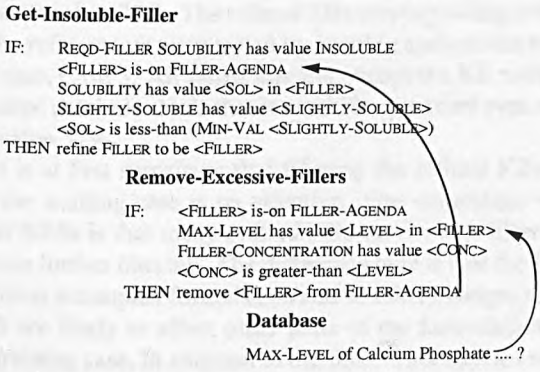

Figure 4. Rule chain for database repair

The remainder of **Remove-Excessive-Fillers** requires the MAX-LEVEL of Calcium Phosphate to be less than the value of FILLER-CONCENTRATION; the value of FILLER-CONCENTRATION can be determined from the trace. KRUST now uses an inductive approach to provide a "best guess" for MAX-LEVEL by using further constraints from a relevant selection from all the training cases:

Positive Examples – training cases (including the current training case) for which TFS wrongly recommends Calcium Phosphate and fails **Remove-Excessive-Fillers** at the same point

Negative Examples – training cases for which TFS correctly recommends Calcium Phosphate and fails **Remove-Excessive-Fillers** at the same point.

The values of FILLER-CONCENTRATION for the positive examples provide a set of upper bounds for the MAX-LEVEL of Calcium Phosphate and KRUST selects the least upper bound. Conversely, the values of FILLER-CONCENTRATION for the negative examples provide a set of lower bounds, and hence a greatest lower bound. The greatest lower bound and least upper bound delimit the new database fact. In this example the most conservative change uses the least upper bound; the most radical change uses the greatest lower bound. Alternatively, the expert could select a value between these two bounds.

4.4.2. Fixing algebraic formulae

TFS contains several algebraic formulae. Again, KRUST has always been able to **identify** rule elements comprising arithmetic calculations as potential faults but has not repaired them. Our experience of these faults in TFS has revealed 2 types. Firstly the conditions under which a formula is applied may be wrong. These faults are repaired with standard KRUST refinement operators including the inductive operator described above (Section 4.4.1.). Secondly, the algebraic formula itself may be wrong. KRUST identifies these faults but currently does not discover a new formula. The expert may wish to volunteer useful information (a new formula or the general format for the formula) but we plan to use positive and negative examples as above to determine a set of points on the curve and then apply a standard curve fitting algorithm.

5. Experience of KRUST with TFS

In this section we describe the faults that have been repaired in TFS. We concentrate on real knowledge engineering issues – the faults occurred in an early version of a real industrial application and the application was modified later in reaction to a real change in company policy. We are therefore not particularly concerned with providing results of the form: "if you randomly select T% of the available data as training data then on average the improved accuracy of the KBS is A% on the (100-T)% remaining data compared to P% prior to learning". We feel that the results here are more important for practising knowledge engineers, since they demonstrate KRUST's effectiveness in a real situation.

5.1. Debugging results

In this section we describe KRUST in its debugging role when it attempts to replicate the manual refinement that has

taken place historically developing TFS-1B from TFS-1A. KRUST refines an early (buggy) version TFS-1A and the training cases are provided by a later (assumed correct) version, TFS-1B. Thus TFS-1B, rather than an expert, is playing the part of an oracle. KRUST has access to the knowledge in TFS-1A, the problem-solving of TFS-1A (in the form of traces), and training cases consisting of specifications and formulations produced by TFS-1B. KRUST has no access to traces or knowledge from TFS-1B.

We generated 208 requirements spread evenly through 13 drugs, 10–500mg doses and requiring 1 or 2 fillers. A hierarchical clustering algorithm applied to the TFS-1A traces for these inputs identified 3 types of fault.

Fault 1: incorrect filler The refinement run produced by Case 5 is typical.

Case 5 Input:	Drug-A, 110mg, 1 filler
TFS-1A Output:	Filler Calcium Phosphate 57.3%
TFS-1B Output:	Filler Calcium Carbonate 58.5%

The blame assignment and refinement generation stages produced 61 potential refinements, one of which identified the database value for the MAX-LEVEL of Calcium Phosphate as missing (Example in 4.4.1.). A set of training cases produced the range [12%, 57%] for MAX-LEVEL. The conservative repair, MAX-LEVEL=57% was overridden by the expert's value, 30%. The refined KBs corresponding to each of the refinements were tested by running each on the training case, Case 5. All failed this test except the KB with the repaired database. Note that it would have passed even with the value 57%.

It is at first surprising that filtering the refined KBs using the training case is so effective. Our experience with other KBSs is that many refined KBs survive this filter and require further filtering. The difference here is that the KBS output is a complex formulation and so other changes to the KBS are likely to affect other parts of the formulation for the training case, in addition to the filler. This appears to be an advantage of refining a design system and is in contrast to experience with diagnostic systems, where one difficulty is selecting relevant training cases with which to filter effectively the many refined KBs generated [12].

Fault 2: incorrect quantity of binder Cases 1 and 52 are examples.

Case 1 Input:	Drug-A, 10mg, 1 filler
TFS-1A Output:	Binder Gelatin 4.1%
TFS-1B Output:	Binder Gelatin 2.1%
Case 52 Input:	Drug-G, 210mg, 2 fillers
TFS-1A Output:	Binder Maize-Starch 4.0%
TFS-1B Output:	Binder Maize-Starch 2.0%

The blame assignment and refinement generation stages produced 93 potential refinements for Case 1. One refinement corrects the fault shown in Figure 5 as follows.

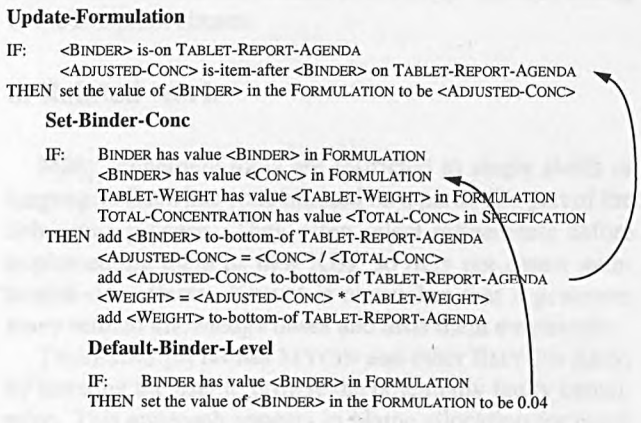

Figure 5. Rule chain for binder level repair

The binder concentration in the FORMULATION will be correctly set from the TABLE-REPORT-AGENDA in **Update-Formulation**, if the ADJUSTED-CONC calculation in **Set-Binder-Conc** is altered. One way of changing this calculation is to alter the value of <CONC>, which was added to the formulation in the conclusion of **Default-Binder-Level**. The back-propagated repair changes the conclusion of **Default-Binder-Level** from 0.04 to 0.02. Again, this repair is the only refined KB to answer the training case, Case 1, correctly.

Fault 3: multiple specification faults Case 183 is typical.

Case 183 Input:	Drug-A, 360mg, 1 filler	
Output:	TFS-1A	TFS-1B
Target Tablet Weight	400mg	450mg
Drug Concentration	9/10	4/5
Filler Concentration	0.0	0.1

... various other discrepancies

The fault dependency mechanism selected Target Tablet Weight as the independent fault. Moreover, there is only one rule that determines TARGET-TABLET-WEIGHT (in fact, it is responsible for all multiple faults):

1st-Guess-Weight

IF DRUG has value <D> in FORMULATION
AND <D> has value <DOSE> in FORMULATION
AND $\text{<WEIGHT>} = \text{round5}\left(\frac{\text{<DOSE>}}{0.1+0.00221\text{<DOSE>}}\right)$
THEN set value of TARGET-TABLET-WEIGHT in SPECIFICATION to be <WEIGHT>

This is a wrong-fire rule whose action must be corrected. Refinements are generated that make this rule fire

correctly; i.e. by setting the correct value for TARGET-TABLET-WEIGHT. The value in <WEIGHT> can be changed by altering the formula, or finding a different value for <DOSE>, or retrieving a different <D> and hence <DOSE>. However, the drug and dose in the formulation are fixed by the input to TFS so cannot be altered. Therefore the only refinement generated is the instruction to repair the algebraic formula. The source of the error has been identified and the proposed curve fitting mechanism (Section 4.4.2.) will implement a repair.

5.1.1. Reaction of the "experts"

Our domain expert has been surprised and encouraged by the success of these refinements. He believes that identifying the missing database entry in Fault 1 and the faulty formula for Fault 3 is very helpful for an expert or knowledge engineer and would be very willing to supply the actual database value or propose a suitable formula for these potential faults.

Another "expert" is TFS-1B; in these experiments its knowledge and formulations take the place of the expert. We have access to the TFS-1B KB and can compare our refined KBs with it. For the Wrong Filler and Wrong Binder Level faults, the repairs that KRUST suggested coincide with the manual updates. For the training case with multiple faults, TFS-1B reveals an additional 1st-Guess-Weight rule for high values of dose; curve fitting would approximate the divergent (cubic) part of the formula.

An inspection of the differences between TFS-1A and TFS-1B revealed that clustering the 208 examples had highlighted all the TFS-1A faults. Therefore the runs above find all the errors. Although it may appear that the repairs are based on single training cases, this is not true; inductive operators require a set of positive and negative training cases.

5.2. Maintenance experience

Our more recent, and incomplete, work investigates the role of refinement for the more difficult maintenance task caused by the change of formulation policy. Now, TFS-1B is the KBS to be refined and TFS-2 is the oracle. We have data from TFS-2 for the original 208 training cases, and we use TFS-1B knowledge and its traces for these cases.

Clustering the TFS-1B cases has identified a class that reveals further debugging changes to the calculation of TARGET-TABLET-WEIGHT. The conditions under which the two TFS-1B formulae apply have changed again. The inductive method in Section 4.4.2. successfully moves the boundary separating the two rules.

Some of the remaining faults found in training cases are due to the paradigm shift. We have been given additional knowledge that supports the changes: excipients have been grouped into 3 classes and excipients in one class are preferred to those in subsequent classes. This is precisely the type of knowledge that KRUST can use with inductive operators to create versions of rules with new conditions relating to the excipient classes.

6. Related work

Many refinement tools are restricted to single shells or languages. Each has concentrated on a particular part of the debugging process. They often select refinements before implementing them as new KBs, so may not detect unintended side-effects. KRUST is unusual in that it generates many refined knowledge bases and tests them empirically.

TEIRESIAS [5] refines MYCIN and other EMYCIN KBSs by assisting the user to browse the potentially faulty knowledge. This approach appears in blame allocation for many more recent, more automated, refinement tools. EITHER [11], FORTE [14], and many other theory revision tools coming from the Machine Learning community, revise Prolog theories. They rely on induction from many examples to replace knowledge that has been removed by their only other refinement operators, rule and rule element deletion. MOBAL [9] acquires and refines knowledge for its own development tool. It contains many knowledge acquisition tools with access to a wide range of knowledge sources. SEEK [7] refines EXPERT KBSs, exploiting the fundamentally numeric format of the rules and conflict resolution strategy, to apply an incremental scoring for potential faults and suitable repairs.

CLIPS-R [10] refines the forward-chaining rules of CLIPS KBSs. Like KRUST it identifies faulty rule elements by working backwards from observed faulty behaviour. But it differs from KRUST in a number of ways. It requires traces containing more information than PFES traces and uses a variety of user-supplied constraints on the correct behaviour of the KBS. It has been applied only to diagnostic systems and it might have difficulty with the generate-and-test mode of design KBSs; e.g. it groups traces sharing an initial sequence of rule firings but these might be overwhelming for a design system.

ODYSSEUS [17] refines Minerva KBSs. A distinctive feature of ODYSSEUS is that the control is explicitly represented as knowledge, and meta-rules allow it to reason about the control strategy and guide the learning of new domain rules. Wilkins' concentration on control supports our belief that inference is an important consideration for refinement, despite it being largely ignored by many refinement tools. Wilkins has also found that specifying meta-level repairs for a **small** set of meta-rule faults is sufficient to solve many refinement problems.

We are lucky to have access to a real KBS application on which to evaluate our refinement tool, especially one con-

taining an archive of versions covering both debugging and maintenance. Many refinement tools rely on corrupted versions of toy applications or simple systems induced from standard machine learning datasets. In particular, it is common for these to have an ample supply of test cases. But in practice, test cases are often scarce and so the field of KBS validation has seen various tools capable of generating test cases with a higher chance of finding faults; e.g. SYCOJET [1]. We believe that refinement tools must make the best use of existing test cases as training data, but if necessary suggest new, complementary test cases [12].

7. Conclusions

We have found that our refinement techniques are equally applicable to design and diagnostic KBSs, since there is nothing fundamentally different about the knowledge content and problem-solving, and so our knowledge skeleton and problem graph models apply. The exploratory nature of design KBSs demands inductive operators, so these are being added to KRUST whilst we continue to develop its ability to cope with real-world systems.

KRUST's speculative approach of generating many potential refinements necessitates effective mechanisms to reject unsuitable repairs. This problem is reduced for design KBSs where the solutions are more complex objects than the single conclusion normally associated with diagnostic applications. Design solutions therefore provide many more constraints on which to evaluate repairs. On the other hand, the complexity of a design demands an understanding of dependencies so that the refinement process can concentrate on underlying faults first.

Access to a real application has provided useful insights into real knowledge engineering issues. TFS is one of the few successful tablet formulation systems because knowledge acquisition is difficult for this complex task. Nevertheless, KRUST found it relatively easy to focus on the various faults. It duplicated the manual refinement from TFS-1A to TFS-1B, and is making progress on the refinement to achieve TFS-2. It is unfortunate that the examples of faulty behaviour that inspired the manual refinement are no longer available, but we use only a few of the abundant test cases generated by our oracles, the various TFS versions.

Acknowledgements

We thank Paul Bentley, Logica, for his assistance with the PFES software interface.

References

[1] M. Ayel and L. Vignollet. SYCOJET and SACCO, two tools for verifying expert systems. *International Journal of Expert Systems: Research and Applications*, 6(3):357–382, 1993.

[2] R. Boswell, S. Craw, and R. Rowe. Knowledge refinement for a design system. In *Proceedings of the European Knowledge Acquisition Workshop (EKAW97)*. Springer, 1997. To Appear.

[3] L. Carbonara and D. Sleeman. Improving the efficiency of knowledge base refinement. In L. Saitta, editor, *Machine Learning: Proceedings of the Thirteenth International Conference*, pages 78–86, Bari, Italy, 1996. Morgan Kaufmann.

[4] S. Craw and P. Hutton. Protein folding: Symbolic refinement competes with neural networks. In A. Prieditis and S. Russell, editors, *Machine Learning: Proceedings of the Twelfth International Conference*, pages 133–141, Tahoe City, CA, 1995. Morgan Kaufmann.

[5] R. Davis. Teiresias. In A. Barr and E. A. Feigenbaum, editors, *The Handbook of Artificial Intelligence Volume 2*, pages 87–101. Morgan Kauffman, Los Altos, CA, 1982.

[6] J. Frank, B. Rupprecht, and W. Schmelmer. Knowledge-based assistance for the development of drugs. *IEEE Expert*, 12(1):40–48, 1997.

[7] A. Ginsberg, S. M. Weiss, and P. G. Politakis. Seek2: A generalized approach to automatic knowledge base refinement. In *Proceedings of the Ninth IJCAI Conference*, pages 367–374, 1985.

[8] T. Menzies. Applications of abduction: Knowledge level modelling. *International Journal of Human-Computer Studies*, 45(3):305–335, 1996.

[9] K. Morik, S. Wrobel, J.-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning*. Academic Press, London, 1993.

[10] P. M. Murphy and M. J. Pazzani. Revision of production system rule-bases. In W. W. Cohen and H. Hirsh, editors, *Machine Learning: Proceedings of the Eleventh International Conference*, pages 199–207, New Brunswick, NJ, 1994. Morgan Kaufmann.

[11] D. Ourston and R. Mooney. Theory refinement combining analytical and empirical methods. *Artificial Intelligence*, 66:273–309, 1994.

[12] G. J. Palmer and S. Craw. The selection of training cases for automated knowledge refinement. In J. Vanthienen and F. van Harmelen, editors, *Proceedings of the 4th European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV97)*, pages 205–215, Leuven, Belgium, June 1997.

[13] A. D. Preece, R. Shinghal, and A. Batarekh. Principles and practice in verifying rule-based systems. *Knowledge Engineering Review*, 7(2):115–141, 1992.

[14] B. L. Richards and R. J. Mooney. Refinement of first-order horn-clause domain theories. *Machine Learning*, 19(2):95–131, 1995.

[15] R. Rowe. An expert system for the formulation of pharmaceutical tablets. *Manufacturing Intelligence*, 14:13–15, 1993.

[16] J. Turner. Product formulation expert system. *Manufacturing Intelligence*, 8:12–14, 1991.

[17] D. C. Wilkins. Knowledge base refinement as improving an incorrect and incomplete domain theory. In Y. Kodratoff and R. S. Michalski, editors, *Machine Learning Volume III*, pages 493–513. Morgan Kaufmann, San Mateo, CA, 1990.