Representing Problem-Solving for Knowledge Refinement

Susan Craw and Robin Boswell
School of Computer and Mathematical Sciences
The Robert Gordon University
Aberdeen AB25 1HG, UK
Email: s.craw,rab@scms.rgu.ac.uk

Abstract

Knowledge refinement tools seek to correct faulty knowledge based systems (KBSs) by identifying and repairing potentially faulty rules. The goal of the KRUSTWorks project is to provide a source of refinement components from which specialised refinement tools tailored to the needs of a range of KBSs are built. A core refinement algorithm reasons about the knowledge that has been applied, but this approach demands general knowledge structures to represent the reasoning of a particular problem solving episode. This paper investigates some complex forms of rule interaction and defines a knowledge structure encompassing these. The approach has been applied to KBSs built in four shells and is demonstrated on a small example that incorporates some of the complexity found in real applications.

Introduction

Knowledge refinement tools support the development and maintenance of knowledge-based decision-support systems by assisting with the debugging of faulty knowledge based systems (KBSs) and the updating of KBSs whose application environment is gradually changing, so that the knowledge they contain remains effective and current.

Many knowledge acquisition and refinement systems have been developed for particular expert system shells (Murphy & Pazzani 1994) or logic programming languages (Richards & Mooney 1995; Ourston & Mooney 1994), or even specific applications. In contrast, we aim to develop a refinement framework that defines a set of generic KBS concepts and refinement steps. We are also implementing an extensible toolkit that provides refinement components to achieve these steps. To this end we are designing an internal knowledge structure that will allow the core knowledge refinement algorithm to reason about the problem-solving in the actual KBS.

We have found that, although knowledge refinement alters the knowledge content of the KBS, the knowledge refinement process must also take account of the KBS's problem-solving; i.e. refinement is concerned with the inferences of the KBS as well as its static knowledge. In this paper we focus on the knowledge structure that represents individual problem-solving events. To achieve our goal of generality, the structure we propose must be sufficiently expressive and flexible to cover the different inference mechanisms found in a variety of KBSs.

We first describe the KRUSTWorks framework and the requirements of the refinement tools it creates. We then explore the effects on inference of chaining direction and conflict resolution, in order to justify our claim that refinement tools must alter their process depending on the inferencing that the KBS applies. This investigation highlights the key features that must be incorporated in the generalised reasoning knowledge structure. Finally we describe the problem graph, the way it is created from a problem solving episode, and how it provides the knowledge about the reasoning that is necessary to suggest useful refinements.

A Generic Refinement Framework

The goal of the KRUSTWorks framework is to provide facilities to allow the construction of a specialised refinement tool, a KRUSTTool, tailored to suit the particular KBS being refined. Central to our approach is the idea that refinement is achieved by selecting appropriate refinement operators from a repository and applying them with a core refinement algorithm to internal representations of the KBS to be refined (Figure 1). The core refinement algorithm is abstracted from experience with specific KRUSTTools applied to KBSs on various platforms (Craw & Hutton 1995; Craw, Boswell, & Rowe 1997): Prolog applies backward chaining, both Clips and Logica's PFES use exclusively forward-chaining rules, and IntelliCorp's PowerModel permits both forward and backward chaining.

KRUSTTools apply the standard refinement steps of allocating blame to potentially faulty rules and then proposing repairs to rules that prevent this faulty behaviour. But KRUSTTools are unusual in generating many refinements, and postponing the selection of which repair to choose until the refined KBSs have been evaluated by executing in the original KBS format on further examples. Figure 2 illustrates the compo-

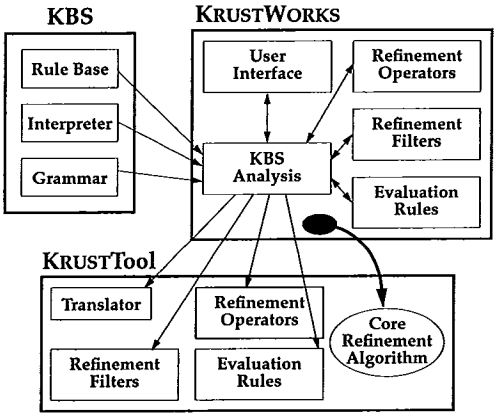

Figure 1: Creating a KRUSTTool from KRUSTWorks

nents of a KRUSTTool, in particular the two key generic knowledge structures, the knowledge skeleton and problem graph, that are central to this approach.

The *knowledge skeleton* is an internal representation of the rules in the KBS. It contains the essential features of the knowledge content; i.e. the KBS' knowledge that is relevant to the refinement process. During translation, each knowledge element is classified within a knowledge hierarchy and this classification is used to reference suitable refinement operators. Details of the knowledge skeleton and its use appear in (Boswell & Craw 1999).

The *problem graph* is the focus of this paper. It captures the problem-solving for the refinement case and allows the KRUSTTool to reason about the fault that is being demonstrated. We investigate a range of KBS inference mechanisms in order to define the problem graph, a general knowledge structure that effectively represents the way rules interact. Minor deficiencies of the problem graph do not cause serious problems for the refinement generation process, given the KRUSTTool's approach of generating large numbers of refinements.

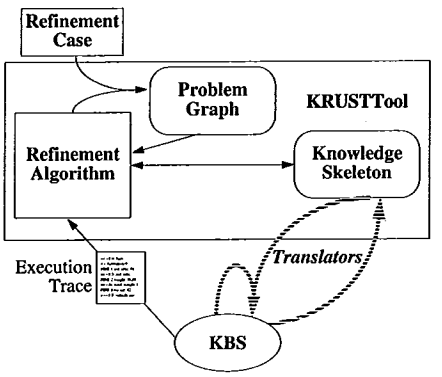

Figure 2: The KRUSTTool and KBS Processes

Reasoning in KBSs

Many refinement systems assume that the KBS employs an exhaustive, monotonic reasoning (Richards & Mooney 1995; Ourston & Mooney 1994). However, real-life KBSs frequently employ non-logical features to restrict the search and exploit non-monotonic reasoning. Since our goal is to refine a wide range of KBSs, we are concerned with the effects of different forms of reasoning on refinement.

A General View of Inference

We start by adopting a simple iterative algorithm (Fensel 1997) as the basis for inference:

1. Initialise the current state of the problem solving
2. Repeat until the termination condition is achieved:
   (a) form the conflict set: match the current state of the problem-solving with rules in the KBS
   (b) apply the conflict resolution strategy: select the rule in the conflict set that should be used
   (c) fire the rule: apply the selected rule to update the current state of the problem-solving.

The initialisation step (1) sets the context of the new problem-solving by representing the features of the new problem and possibly indicating the goal to be achieved. Therefore knowledge refinement is not concerned with this problem initialisation step.

In contrast, the iterative loop (2) determines the reasoning, and knowledge refinement seeks to alter the knowledge that is applied here. Identifying the conflict set (2a) is a purely logical step and so refinement involves altering the knowledge in rules that are, or should be, in the conflict set. All refinement systems apply this type of change.

However, many refinement systems assume an exhaustive backward chaining approach in which conflict resolution (2b) is simply a search ordering mechanism and has no ultimate effect on the solutions proposed by the KBS. This assumption is appropriate for Prolog-like applications with no non-logical features, but is not true for real-life applications that commonly employ mechanisms, such as rule priority and conflict resolution, to restrict the search for a solution. This is particularly true when the reasoning is forward chaining, and is relevant if the termination condition determines that the reasoning halts as soon as the goal is proved.

Firing the selected rule (2c) can be achieved in different ways. Monotonic systems add the new knowledge irrespective of the existing knowledge already inferred. However many systems are non-monotonic in that knowledge can be overwritten (e.g. the field of an object) or the old fact is retracted and a new one asserted. Such behaviour is unusual in Prolog applications but common with Clips facts, or PowerModel objects. These effects are considered in the next section.

In our diagrams that illustrate reasoning we adopt the convention that leaf nodes at the bottom of the graph are observable facts, nodes at the top of the graph

are the goals of the reasoning process, and rules at the top of the graph are called *end-rules*. We shall also be concerned with the order of rule firings; this is shown in a left to right ordering of branches from a rule node.

When Chaining Direction Matters

In KBSs which exhaustively find all solutions and are monotonic, backward and forward chaining find exactly the same rule chains and so refinement of each system is equivalent. We now investigate how selection and non-monotonicity complicate forward chaining and the circumstances in which chaining direction affects the conclusion of the KBS.

Selection from the conflict set alters the order in which inferences are made. Although on many occasions this is used simply to guide the reasoning towards more efficient routes and so has no effect on the ultimate solutions, under some circumstances, such as early termination, the order in which solutions appear is important because some are ignored.

A simple example when backward and forward chaining differ occurs when the KBS's inference terminates as soon as the first end-rule fires. Figure 3 contains a small example. The rule priority is shown in square brackets and rules with higher priority are fired first. If we assume P and Q are true, under forward chaining, R2 fires first, then R4, and the conclusion is X=on. In contrast, under backward chaining, R3 fires first, then R1, and the conclusion is X=off.

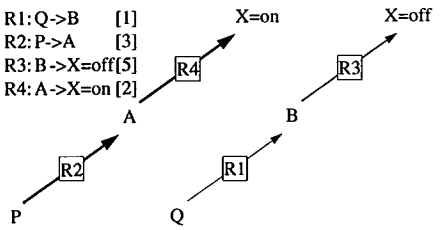

Figure 3: Inference stops when an end-rule fires

Thus, when the relative priority of rules differs between layers of the inference, the priority of the first layer to fire takes precedence; i.e. the leaf rules for forward chaining, and the end-rules for backward chaining. This effect also occurs when there are several *potentially clashing* rules that infer different values for the same attribute without demanding the rules are end-rules, as happened in Figure 3. Examples include Clips rules asserting new ordered terms with the same first element, Prolog clauses having heads that unify, Clips rules updating multi-valued fields, and PFES rules adding to PFES agendas[1]. In this case, conflict resolution in later cycles is affected by the order in which the various instantiations from previous cycles are retrieved.

[1]A PFES agenda is a stack or queue of potential formulation components. This should not be confused with the Clips execution agenda; i.e. the ordered conflict set.

Non-Monotonicity is introduced when knowledge is retracted or overwritten during problem solving. Explicit retraction can occur in backward and forward chaining. Forward chaining systems may also update objects, so that each rule in turn overwrites the value written by the previous rule. Again sets of potentially clashing rules are important.

Figure 4 contains another small rule-set where now suppose Temp is an overwriting property. Under forward chaining when P and Q are true, R1 fires first, R2 fires next overwriting Temp=60 with 50 (shown as a dotted arrow), then R4 fires concluding X=off. R3 is not able to fire since R2 fires first, thereby making its antecedent false; this is shown as a shaded area.

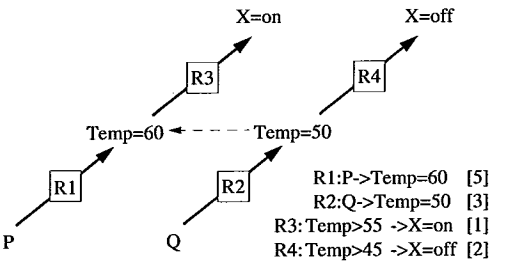

Figure 4: Chaining with Overwriting Rules

Thus, for clashing rules with overwriting conclusions under forward chaining, the lowest priority satisfied path in the graph leads to the eventual solution. The priority of each path is determined by the priority of the leaf node rules. Under backward chaining, exactly the reverse is true; the first rule to fire is the end-rule with the highest priority, here R4 which then chains back to R1 concluding X=off, but this is overwritten with X=on when R3 fires.

This example also demonstrates that the effect of rule priority is not just to choose between the potentially clashing rules R1 and R2 but it also encompasses the rules with which they chain. Thus, rule priority has an influence in a vertical direction, not just horizontally.

Self-Disabling Rules These rules contain exception conditions that ensure only the first from a set of rules fire. They are a specialised form of non-monotonicity since later rules are in effect retracted, and the result is to select only the *highest* priority rules in the direction of chaining. Self-disabling rules constitute a special case of negation and occur frequently in PFES KBSs; e.g. choosing a binder in the tablet formulation application (Craw, Boswell, & Rowe 1997) where the exception checks whether the value of binder has already been set:

```
IF in-agenda stability-agenda
    ?stability after gelatin
AND ?stability greater 90 AND ...
THEN refine-attribute binder gelatin
UNLESS attribute-has-value binder ?any
```

Self-disabling also occurs in backward chaining systems, as with Prolog's cut (!) for commitment; e.g.

```
temp(40) :- p, !.     x :- temp(T), T > 45.
temp(50) :- q.
```

where the first temp rule firing precludes the later temp rule and so the rule for x would not fire.

Conflict Resolution Strategies

We have consistently referred to rule priority and implied that we are restricted to a Clips-like salience attached to static rules. However, Clips uses rule priority only as the first criterion for choosing which rule from the conflict set should fire. Where rule priorities are equal, a variety of further strategies are available; other shells often offer a subset of those available in Clips. The effects of selection and non-monotonicity apply equally to other conflict resolution strategies such as recency, specificity, LEX and MEA[2].

Representing Reasoning

We have seen that the behaviour of forward-chaining rules can be different from that of backward-chaining ones. Therefore, refinement systems need to alter their process depending on the chaining direction of the KBS. We have also seen that the behaviour of forward-chaining rules is often more complex than that of backward-chaining ones, so that if a common procedure is to be adopted for handling both, its form will probably depend largely on the requirements of forward-chaining rules. The comparison of forward and backward chaining identifies the areas of difference and provides a foundation for the generalised mechanism that we propose for both forward and backward chaining KBSs.

Refinement consists of determining what happened, then trying to change it. There are two ways of determining what happened: querying the KBS and then reasoning about rule priorities and conflict resolution in order to *infer* what happened (i.e. running a partial simulation), or else looking at the execution trace to find out what *actually* happened. We have found the retrospective approach works well for backward chaining rules (Craw & Hutton 1995), but the preceding examples suggest that accurately simulating forward-chaining rules is likely to be difficult, so we propose an approach based on traces.

The Problem Graph

We define a knowledge structure that represents the reasoning by capturing both the rules that fired for a given problem-solving activity and the knowledge that might have been applied at each stage in the process. The knowledge is acquired directly from the execution trace and by reasoning about the contents of the trace. Its content informs the blame allocation and refinement generation stages in the KRUSTTool's cycle.

Previously, with backward chaining systems, navigating a simple virtual structure worked well, so we are now extending this approach to forward chaining systems, but here we have found it helpful to create the structure explicitly. We have seen that the ordering of the rules in the conflict set has an effect only when the rules are clashing and so involve some form of negation, retraction, or overwriting. From the point of view of refinement it does not matter why they clash provided we know how to re-order the firing agenda. The problem graph contains a history of the firing agenda and its effects. It links the knowledge that was applied to the knowledge that was inferred, represents relative timings of these events, and identifies knowledge that might have been applied. The problem graph in Figure 5 is applied in a later example.

The *positive* part of the problem graph is extracted directly from the execution trace and contains the knowledge that has been applied. *Rule activation nodes* maintain logical knowledge such as the variable bindings associated with this activation, and timing information about when the rule was activated, when it was fired, and when it was retracted, if appropriate. Similarly, a *fact assertion node* stores the time of assertion and the time of retraction if appropriate. The arcs in the graph link rule activations to the facts they conclude, and fact assertions to the rule activations whose antecedents they match. When an antecedent does not involve another rule (e.g. an inequality check), the structure records the test's truth as *T* or *NIL*.

We have been discussing rule activations and fact assertions that actually took place when the KBS was run on the refinement example, but this is not enough. A problem graph also has a *negative* part consisting of rules activations and facts which did not appear during the run, but which could contribute to the desired behaviour of the system. We refer to these as non-activations and non-facts. Some of a non-activation's antecedents may actually be satisfied; these are connected to facts as in the positive part. Those that fail are recorded as *failure links* connecting each antecedent to matching non-facts.

The problem graph representation also applies to *negated* antecedents; we assume negation as failure in a closed world. For negated knowledge components, we must decide how to link facts and non-facts. If `not(Goal)` is satisfied then there is a link to *T*. Conversely, if `not(Goal)` fails then there is a failure link from the offending fact, `Goal`. When the negated antecedent does not involve other rules, then the truth of the antecedent as a whole is recorded; e.g. if the antecedent `not(X < 42)` fails, then *NIL* is recorded.

Building the Problem Graph

The positive part of the problem graph is derived directly from the KBS's execution trace by extracting information about rule firings and their associated vari-

[2]LEX and MEA extend recency by considering the relative assertion times of facts satisfying individual antecedents.

able bindings, and when facts were asserted and retracted. The negative part of the problem graph is formed by exhaustively exploring backwards from the desired conclusion to identify potential routes through the knowledge. When the KBS itself is backward chaining then both steps can be naturally combined. This may explain why most current work on refinement has dealt only with backward-chaining rules. Refining forward chaining rules is more complicated since they execute in the opposite direction to the refinement process, and so the KRUSTTool must find a way to ensure that the chosen route found by backward chaining is actually executed under the KBS's forward chaining control.

A two-stage translation process creates the positive part of the problem graph from the trace. This is needed since there is not a 1-1 relationship between nodes in the problem-graph and lines in the original trace. For example, separate rule activation and rule firing statements in the Clips trace contribute to the same rule activation node in the problem graph. Conversely, a rule firing in the trace may contribute information to both a rule activation node and a fact node. The translation is therefore most conveniently done by first creating an internal representation of the trace, and then manipulating this internal trace to construct the positive part of the problem graph.

Furthermore, representing rule firings themselves is not sufficient. We also need to know the content of working memory before any rule firing, to determine whether a rule failed to fire because its antecedents were not satisfied, or because its priority was too low. Many KBSs have tracing facilities that provide all this information; e.g. for a Clips trace we elect to watch rules, facts and activations, with PowerModel we select the highest level of verbosity. Otherwise, it is often possible to deduce a snapshot of working memory from previous rule firings; we use this with PFES.

The negative part of the problem graph is constructed by backward chaining from the desired conclusion in the knowledge skeleton. First, the desired conclusion is created as a non-fact. Then rule non-activations are created for all the rules that could conclude this fact, with appropriate bindings. The algorithm then proceeds recursively: for each rule antecedent in a non-activation, if the antecedent is not satisfied by an existing fact, then a matching non-fact is created. The procedure terminates when it reaches non-facts that do not match the conclusion of any rule (observables).

If a rule's antecedents are matched by multiple facts or non-facts, then non-activations are created for all the possible combinations, provided that the variable bindings are compatible; for example, if one antecedent is matched by one fact and one non-fact, and another by two non-facts, then up to four rule non-activations will be created.

Problem Graphs in Practice

We have built problem graphs from Clips, PowerModel and PFES KBSs and their traces. Both stages of the translation from trace to positive problem graph, together with the first internal representation of the trace, are specific to the particular KBS. The form of the problem graph itself is generic. Consequently, the creation of the negative part of the problem graph is independent of the KBS.

Rather than demonstrating the approach using a complex KBS, we have chosen to illustrate the process on a very simple Clips KBS:

Initial Facts: (c), (a), (b)

Rules with Salience 5	Rules with Salience 10
rule1: (a)	rule4: (not (result ?)), (p), (r)
⇒ (assert (p))	⇒ (assert (result x))
rule2: (b)	rule5: (not (result ?)), (q), (r)
⇒ (assert (q))	⇒ (assert (result y))
rule3: (c)	
⇒ (assert (r))	

Suppose also Clips applies the LEX conflict resolution strategy when rule salience does not distinguish a single rule on the activation agenda. This generates the Clips trace showing the facts being asserted one at a time, each followed by a Clips activation when the rule it satisfies is placed in the conflict set:

```
==> f-1     (c)
==> Activation 10     rule3: f-1
==> f-2     (a)
==> Activation 10     rule1: f-2
==> f-3     (b)
==> Activation 10     rule2: f-3
FIRE    1 rule2: f-3
==> f-4     (q)
FIRE    2 rule1: f-2
==> f-5     (p)
FIRE    3 rule3: f-1
==> f-6     (r)
==> Activation 5      rule5: ,,f-4,f-6
==> Activation 5      rule4: ,,f-5,f-6
FIRE    4 rule4: ,,f-5,f-6
==> f-7     (result x)
<== Activation 5      rule5: ,,f-4,f-6
```

LEX fires the rules as rule2, rule1, rule3, since the most recently asserted fact is used first. To discriminate between rule4 and rule5, since they are both activated simultaneously (when fact r was asserted), LEX uses the next most recent pairs of antecedents, p and q. p was asserted more recently, so rule4 fires. But when rule4 fires this removes rule5 from the conflict set (indicated by <==) and so Clips's solution is result(x). Notice that negation is not explicitly mentioned in the trace but the knowledge skeleton provides the information.

Suppose in this example that the "correct" solution is known to be result(y). Figure 5 shows the problem graph that is constructed. Rule activations are rectangular nodes, fact assertions are oval nodes, and dashed outlines indicate non-facts and non-activations. Arcs indicate the various links; conjunctive antecedents are shown circled, arrows point towards conclusions, fail-

ure links are shown dashed, and negations are indicated with a ⋈ through the link.

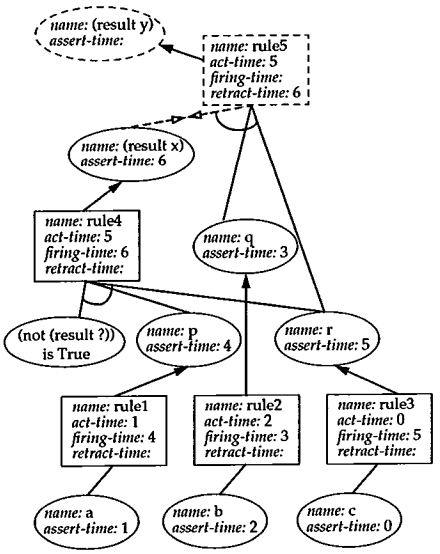

Figure 5: A Sample Problem Graph

One of the many features of PowerModel and PFES which makes the construction of the problem graph more difficult is the phenomenon of backtracking: whenever a rule is executed, the interpreter forces the generation of all possible variable instantiations, rather as if a Prolog goal were followed by a `fail` statement. Consequently, each successful rule firing requires the creation of a separate rule activation node, and determining the variable bindings for each rule firing may require some searching forwards and backwards through the trace.

Refinement Generation

Blame allocation and refinement generation seek to disable a faulty proof and to identify and enable a correct proof in the rules. The problem graph contains the faulty proof and potential proofs together with the information that is required for the KRUSTTool to find a way to ensure that the chosen route is actually executed.

The most common type of refinement operators found in refinement systems are those that alter the logical content of the rule and so affect whether the rule is added to the conflict set. However, we believe there is another important class of refinements: those that alter the way the rule is handled by the conflict resolution strategy and so affect whether the rule is selected to fire or not. This is particularly relevant for forward chaining rules where selection plays a larger role.

To correct the behaviour of the KBS, it may be necessary to make several changes at the same time. For example, if the KRUSTTool wants a rule to fire, but two of its antecedents are currently unsatisfied, the KRUSTTool will probably have to make at least two changes somewhere in the rules. We describe all the individual changes required to fix a particular fault as a refinement. The output from the algorithm is a series of refinements, each of which individually is designed to correct the behaviour of the KBS.

Logical Refinements

This type of refinement is common to all refinement systems, although the changes that achieve the refinement can vary. Here we describe how a KRUSTTool generates these from the problem graph.

To *enable a desired conclusion* the KRUSTTool enables any one rule which matches that conclusion. Such rules may be read from the problem graph, where the desired fact is linked to one or more rule non-activations. To enable a rule, for each failed antecedent in the associated non-activation, the KRUSTTool either weakens that antecedent so that it is satisfied for the refinement case, or else applies the algorithm recursively, by enabling a non-fact linked to the failed antecedent.

To *prevent an undesired conclusion* the KRUSTTool disables all rules that fired and match that conclusion. To disable a rule, the KRUSTTool disables any one of its antecedents, or deletes the rule. It disables an antecedent either by strengthening it, or by disabling all the rules linked to the fact that caused the antecedent to be satisfied.

The presence of negation requires the following extensions to the above basic algorithms. To *enable a failed negation*, the KRUSTTool identifies the matching fact in the problem graph, and disables all the rule activations that concluded that fact, using the algorithm above. Conversely, to *disable a negation*, the KRUSTTool identifies the matching non-fact, and enables any one of the rule non-activations which conclude that fact.

Conflict Resolution Refinements

It is always possible to generate only logical refinements but in certain circumstances it is appropriate to also generate refinements based on the conflict resolution strategy. Conflict Resolution is used as a basis for refinement in two situations: when a rule is activated but then de-activated before firing, and when a group of potentially clashing rules fires in the wrong order, so that, for example, the desired conclusion is overwritten by a later conclusion. Multiple potentially clashing rules on an agenda would most naturally arise in the case of a group of self-disabling rules.

For the purposes of refinement, the two situations are handled in the same way: we observe that rule R1, say, fires before R2, and we wish to ensure that R2 fires before R1. One way to achieve this is by applying a logical refinement to disable rule R1. However, when the KRUSTTool determines from the problem graph that R1 and R2 are on the rule execution agenda at the same time then an alternative refinement modifies the priority of R2 so that it is higher than R1. To determine the

required priority change, it is also necessary to identify any other rules present on the agenda at the same time which would also prevent R2 from firing.

These refinements have assumed a conflict resolution strategy based on rule priority, but other strategies are possible. In Clips, any conflict resolution refinement can be applied as a salience change, because Clips uses other strategies only when rule priorities are tied. However, in some situations rule priority changes might be too radical and this might prevent the KRUSTTool from finding a satisfactory refinement. We now consider which conflict strategy refinements should be adopted. Firstly, the KRUSTTool should *not* refine the strategy itself, on the grounds that such a global change would be non-conservative, and would have unpredictable side-effects. Secondly, the KRUSTTool does *not* propose to add or delete antecedents *simply* in order to modify the effects of a specificity/generality strategy, again because of undesired side-effects. The remaining strategies are based on the order of fact assertions, and it is possible to take these into account when generating refinements. The task of altering the firing order of rules at one level can be carried out by a recursive procedure which alters priorities either at that level, or at lower levels. Sucn lower level refinements affect the order of fact assertions, and and hence of rule firings, at higher levels. The example in the next section illustrates this.

Refinement Using a Problem Graph

We have applied KRUSTTools to a range of KBSs, and the problem graph is simply a common format in which to represent the various reasoning mechanisms that can happen in KBSs. The ability of the core refinement algorithm to generate useful refinements has been demonstrated elsewhere: a KRUSTTool achieves competitive results with other refiners when applied to a benchmark backward-chaining Prolog KBS (Craw & Hutton 1995); a PFES KRUSTTool debugs an early version of Zeneca's Tablet Formulation System, implemented in forward-chaining PFES, (Craw, Boswell, & Rowe 1997).

Our problem graph approach works well for larger KBSs but these are difficult to illustrate. Instead, we revert to the simple problem graph in Figure 5 as our example and illustrate the problem graph's use when refining this KBS. The refinements are described below but are highlighted by shading in Figure 6 where disabled rules are shown with "X"s and the changed priorities are shown with lightly shaded arrows.

Three Logical Refinements:

- disable rule1, rule3 or rule4

  Disabling rule3 does not actually repair the rules, as it prevents rule5 from firing as well as rule4, but this is an unpredictable side-effect and is identified by the KRUSTTool when it tests its proposed refinements. In practice each disabling can be implemented in many different ways by disabling any one of the antecedents or deleting the rule.

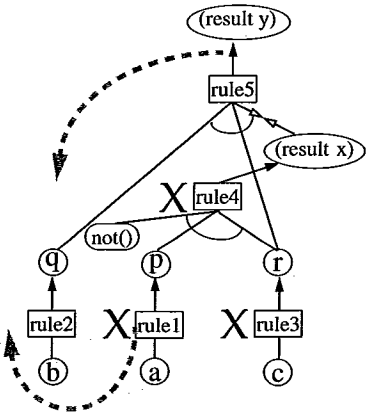

Figure 6: Generated Refinements

Two Conflict Resolution Refinements:

- increase the priority of rule5 above rule4 by setting its salience to 11
- increase the priority of rule1 above rule2 by setting its salience to 6.

  Increasing the priority of rule1 has the effect that rule2's conclusion is asserted later and so LEX will fire rule5 before rule4. The KRUSTTool could in principle recurse a further level when generating these refinements, and change the fact order.

Even this very simple example demonstrates the variety of refinements and the interactions that can occur.

Comparison with Other Approaches

The problem graph we have defined is the basis of a common representation for reasoning. Fensel *et al.* (1997) also seek to define problem solving methods (PSMs) in a task-independent way so that they can re-use the PSMs by tailoring them to a specific task. Fensel *et al.*'s goal and approach is thus similar to KRUSTWorks'. Their adapter selects a PSM whose abilities partially match the task's requirements and specialises the PSM for the task. Similarly, our KRUSTWorks framework allows the use of generic components which then act on a particular KBS

Both Etzioni (1993) and Smith & Peot (1996) create structures similar to our negative problem graph. Since their domain is planning, the nodes correspond to operators and pre-conditions rather than rules and conditions. Their graphs are constructed in a similar way to ours, by backward chaining from the final goal, and are used for reasoning about potential interactions between operators. In both cases, the graph is created before the planner is run, so no structure corresponding to our positive problem graph is created. The purpose of the graphs is to improve the subsequent performance of the planners. Smith & Peot's *operator graph* determines which rule conditions have the potential to lead

to recursion; knowing this, they can prevent the planner from entering an infinite loop. Etzioni uses the *problem space graph* to learn control rules to guide operator application, rejecting inappropriate operators and ordering appropriate ones.

CLIPS-R (Murphy & Pazzani 1994) refines Clips KBSs, and is the only other refinement system for forward-chaining rules. CLIPS-R shares KRUSTWorks' aim of applying knowledge refinement to real-world expert systems, and thus takes account of non-logical features such as rule priority. Instead of using the trace from a single example, CLIPS-R combines the traces of all examples into a trie-structure, grouping together those examples which have a common initial sequence of rule firings. CLIPS-R chooses to refine first the group that exhibits the greatest proportion of errors. The trie-structure is not used directly for refinement, in the way that a KRUSTTool's problem graph is, but CLIPS-R does backward chain from failed goals; it just does not build an explicit structure like KRUSTTools.

Conclusions

Many refinement tools assume an exhaustive backward chaining control and therefore restrict attention to logical refinements. We have found that additional refinements may be relevant; these alter the knowledge content so that the rule is handled differently by the conflict resolution strategy. An exploration of this behaviour showed that backward and forward chaining could produce different answers when the problem-solving terminated prematurely (e.g. as soon as a solution was found) or non-monotonicity was introduced. These are common features of industrial KBSs where default reasoning is standard practice to achieve efficiency.

The blame allocation phase of knowledge refinement naturally reasons backwards from the desired solution. Therefore, forward chaining and mixed chaining KBSs pose some problems. The proposed refinement must ensure that the solution path found by backward chaining is actually executed, and so blame allocation involves more complex reasoning about what will happen.

We defined a knowledge structure that captures the reasoning: what happened and what might have happened. This knowledge does not need to faithfully simulate the reasoning since the KRUSTTool does not rely on it exclusively for generating refinements. Instead, it forms a basis for the reasoning about refinements, but all refinements are checked by executing the refined KBS on its native platform. We have noted that the knowledge cannot be extracted entirely from the execution trace, but requires additional reasoning about the static knowledge.

The approach has been evaluated with forward or backward chaining systems in Prolog, Clips and PFES, and the mixed chaining of PowerModel. Experience with these KBSs has shown that the problem graph contains the reasoning information that allows a KRUSTTool to identify what may have gone wrong and to suggest possible repairs.

Acknowledgements

Susan Craw is currently on sabbatical at the University of California, Irvine. The work described in this paper is supported by EPSRC grant GR/L38387, awarded to Susan Craw. We also thank IntelliCorp Ltd and Zeneca Pharmaceuticals for their contributions to the project and the anonymous reviewer who highlighted the related work from planning.

References

Boswell, R., and Craw, S. 1999. Knowledge Modelling for a Generic Refinement Framework. *Knowledge Based Systems* (in press). Also appears in Proceedings of the BCS Expert Systems Conference, 58–74, Springer, 1998.

Craw, S., and Hutton, P. 1995. Protein folding: Symbolic refinement competes with neural networks. In Prieditis, A., and Russell, S., eds., *Machine Learning: Proceedings of the Twelfth International Conference*, 133–141. Tahoe City, CA: Morgan Kaufmann.

Craw, S.; Boswell, R.; and Rowe, R. 1997. Knowledge refinement to debug and maintain a tablet formulation system. In *Proceedings of the 9TH IEEE International Conference on Tools with Artificial Intelligence (TAI'97)*, 446–453. Newport Beach, CA: IEEE Press.

Etzioni, O. 1993. Acquiring search-control knowledge via static analysis. *Artificial Intelligence* 62:255–301.

Fensel, D.; Motta, E.; Decker, S.; and Zdrahal, Z. 1997. Using ontologies for defining tasks, problem-solving methods and their mappings. In Plaza, E., and Benjamins, R., eds., *Knowledge Acquisition, Modeling and Management, Proceedings of the 10th European Workshop (EKAW97)*, 113–128. Sant Feliu de Guixols, Spain: Springer.

Fensel, D. 1997. The tower-of-adapters method for developing and reusing problem-solving methods. In Plaza, E., and Benjamins, R., eds., *Knowledge Acquisition, Modeling and Management, Proceedings of the 10th European Workshop (EKAW97)*, 97–112. Sant Feliu de Guixols, Spain: Springer.

Murphy, P. M., and Pazzani, M. J. 1994. Revision of production system rule-bases. In Cohen, W. W., and Hirsh, H., eds., *Machine Learning: Proceedings of the 11th International Conference*, 199–207. New Brunswick, NJ: Morgan Kaufmann.

Ourston, D., and Mooney, R. 1994. Theory refinement combining analytical and empirical methods. *Artificial Intelligence* 66:273–309.

Richards, B. L., and Mooney, R. J. 1995. Refinement of first-order horn-clause domain theories. *Machine Learning* 19(2):95–131.

Smith, D. E., and Peot, M. A. 1996. Suspending recursion in causal-link planning. In *Proceedings of the Third International Conference on AI Planning Systems*. Edinburgh, Scotland: AAAI press.