Knowledge Modelling for a Generic Refinement Framework

Robin Boswell and Susan Craw
School of Computer and Mathematical Sciences, The Robert Gordon University
St Andrew Street, Aberdeen AB25 1HG, UK

Abstract

Refinement tools assist with debugging a KBS's knowledge, thus easing the well-known knowledge acquisition bottleneck, and the more recently recognised maintenance overhead. Existing refinement tools are developed for specific rule-based KBS environments, and have usually been applied to artificial or academic applications. Hence there is a need for tools which are applicable to industrial applications. However, it would be wasteful to develop separate refinement tools for individual shells; instead, the KrustWorks project is developing re-usable components applicable to a variety of KBS environments. This paper develops a knowledge representation that embodies a KBS's rulebase and its reasoning, and permits the implementation of core refinement procedures, which are generally applicable and can ignore KBS-specific details. Such a representation is an essential stage in the construction of a generic automated knowledge refinement framework, such as KrustWorks. Experience from applying this approach to Clips, PowerModel and Pfes KBSs indicates its feasibility for a wider variety of industrial KBSs.

Keywords: Knowledge Refinement, Knowledge Representation, Knowledge Acquisition.

1 Introduction

The well-known knowledge acquisition bottleneck encompasses both the original knowledge elicitation, and the debugging of the knowledge while the knowledge based system (KBS) develops. As KBSs become more routinely used in industry, their maintenance becomes a further knowledge management issue. The evolution of methodologies such as KADS [1], organises the knowledge development process, but there is a demand for knowledge refinement tools that assist with the acquisition, debugging and maintenance of the knowledge itself. A knowledge refinement tool assists a knowledge engineer by identifying places where the knowledge may need to be changed. For knowledge acquisition, it identifies potential gaps in the knowledge and incorporates missing knowledge into the KBS. For debugging, it identifies potential faults in the knowledge and suggests possible repairs. In contrast to debugging, knowledge maintenance refines the knowledge because the problem-solving environment has changed in some way; in this case, the knowledge must be updated to match the new environment.

Knowledge refinement tools each perform the same general steps. The tool is presented with a faulty KBS and some evidence of faulty behaviour; often this consists of examples that the KBS fails to solve correctly, together with the correct solutions. The refinement tool reacts to a piece of evidence by undertaking the following three tasks: *blame allocation* determines which rules or parts of rules might be responsible for the faulty behaviour; *refinement generation* suggests rule modifications that may correct the faulty behaviour; and *refinement selection* picks the best of the possible refinements. The goal of refinement is that the refined KBS correctly solves as many of the examples as possible, with the expectation that novel examples will also have an improved success rate.

Most knowledge refinement systems are designed to work with KBSs developed in a single language [2, 3, 4], or a particular shell [5]. However, it is wasteful to develop refinement tools for individual languages and shells. We prefer to investigate re-usable refinement components that can be applied to a variety of KBS environments. This paper concentrates on the knowledge representation issues, and the rest of this section investigates the knowledge demands of the core refinement processes. Section 2 considers how they can be satisfied by generic structures that organise the key roles adopted by components of rules. In Sections 3 and 4 we describe our experience of applying these generic knowledge structures to various KBSs. Section 5 investigates the approaches used by other refinement tools. In Section 6 we draw some conclusions about the usefulness of these structures and their utility in our long term goal of providing a framework of refinement components in the KrustWorks project.

1.1 The Refinement Process of a KrustTool

We base this paper on experience with our Krust refinement system [6]. Figure 1 shows a KrustTool[1] performing the operations highlighted above. The KBS's problem-solving for one training example is analysed, and blame is allocated to the knowledge that has taken part in the faulty solution, or which failed to contribute to the solution as intended. The experiment toolset generates repairs that correct this faulty behaviour. A rule is prevented from firing by making its conditions harder to satisfy, or by preventing rules that conclude the knowledge required by the original rule's conditions. Conversely, a rule is encouraged to fire by making its failed conditions easier to satisfy, or by encouraging rules that conclude the knowledge required by the original rule's failed conditions. Knowledge specific refinement operators implement these repairs on the rules. KrustTools are unusual in proposing many faults and generating

[1]We now refer to refinement tools that apply the basic mechanism of the original Krust system as KrustTools. We are developing a KrustWorks framework (Section 6) from which an individual KrustTool for a particular KBS will be assembled; i.e. there is not one unique KrustTool.

many repairs initially, and so a KRUSTTool applies filters to remove unlikely refinements before any refined KBSs are implemented. It then evaluates the performance of these refined KBSs on the training example itself and other examples that are available. A detailed description of the execution of an early KRUST appears in [7].

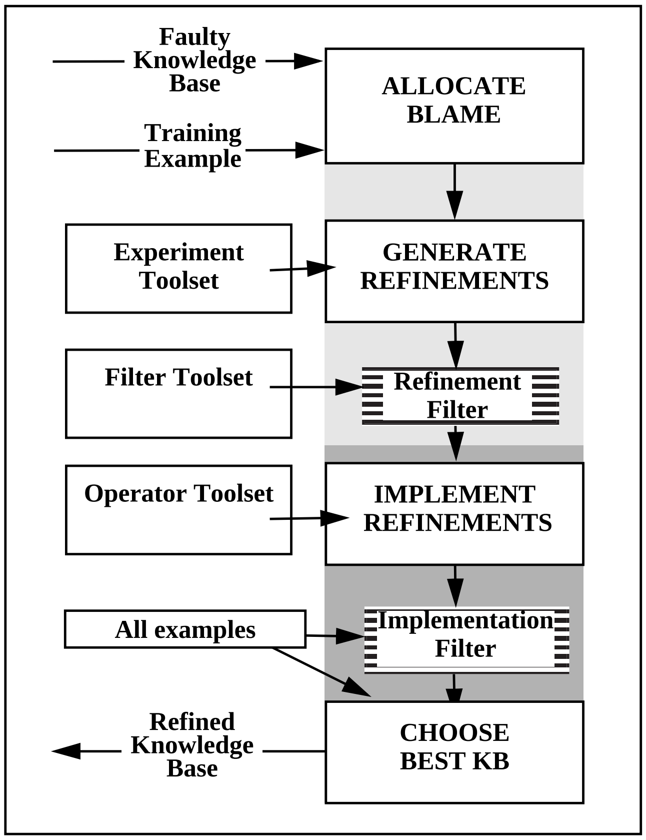


Figure 1: The Operation of a KRUSTTool

1.2 The Knowledge Demands of a KRUSTTool

The refinement process relies on determining what knowledge was applied to solve the given training example and what knowledge might have been applied instead (Blame Allocation), and how to alter the knowledge (Refinement Generation). A refinement tool therefore reasons about actual and potential interactions within the KB, and in particular rule chaining behaviour. An important result of our work on knowledge refinement is the conclusion that the necessary information for the refinement of any KBS may be represented in two structures.

The *Knowledge Skeleton* is an internal representation of the rules in a KBS. The knowledge skeleton allows the refinement tool to determine what knowledge is applied and how rules can chain. The creation of a knowledge skeleton requires the existence of a common knowledge representation language, which can represent any feature in any shell to which KrustWorks is to be applied. The skeleton itself is built by a shell-specific translator. The remaining sections of this paper concentrate on the representation language for the knowledge skeleton and the use of the knowledge skeleton in KrustTools.

The *Problem Graph* represents the KBS's problem-solving for an incorrectly solved example. Details of the problem graph and its use in KrustTools will appear in a later paper. Figure 2 shows a problem graph when the KBS solution is '−' but the correct answer is '+'. The initial facts are shown as circular leaf nodes, provable knowledge is also shown as circles, and square nodes represent knowledge that is currently not provable but would help to correct the error. Each rule is labelled by a diamond, linked to its conditions beneath it and its conclusion above. A circular arc indicates that a rule has two or more conditions forming a conjunction. For example, rule $R7$ has conditions G $\wedge$ H and conclusion B. Rule $R8$ has condition H and conclusion C. The *positive*

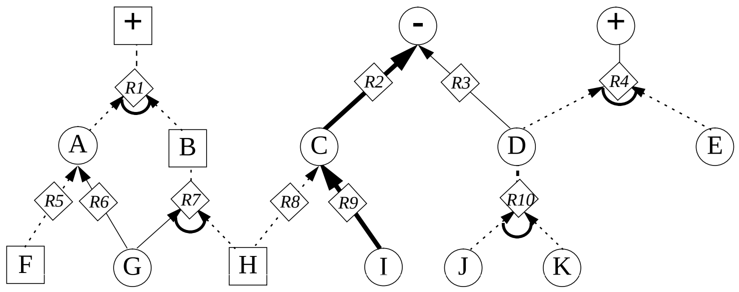


Figure 2: A Sample Problem Graph

part of the problem graph is shown as solid lines and represents the knowledge that was applied during the problem-solving. The *negative* part of the problem graph is shown with dotted lines and highlights those parts of rules which may be changed to deduce the correct answer '+'. The negative part of the graph is constructed with reference to the desired goal and the knowledge skeleton, since it can not be derived from the observed behaviour of the KBS. We represent the control strategy by organising the sub-graphs in a left-to-right order, where the leftmost rules are those chosen earliest for execution. The chain of reasoning that leads to the KBS's conclusion is therefore the leftmost sub-graph containing circled nodes only; nodes are circled because these rules' conditions were satisfied, and they appear as the leftmost of the satisfied rules because they were selected in preference to other satisfied rules. This chain has been highlighted using bolder lines in Figure 2. Blame allocation identifies square nodes leading to '+' that should be altered to allow them to fire (leftmost sub-graph), circular nodes leading to '−' that should be prevented from firing (middle sub-graph), and circular nodes leading to '+' that should be altered to make them more competitive in the control strategy (rightmost sub-graph).

It is important to note that the positive half of the problem graph is derived from the actual observed behaviour of the KB, not from an internal simulation, so is guaranteed correct. On the other hand, the negative half of the graph does require a simulation of potential KB behaviour, and hence may introduce inaccuracies. However, any consequent incorrect repairs will be detected during testing; and if the ideal repair is missed, then the best of the many other refinements that the KRUSTTool generates will be applied instead.

1.3 Using the Knowledge Structures

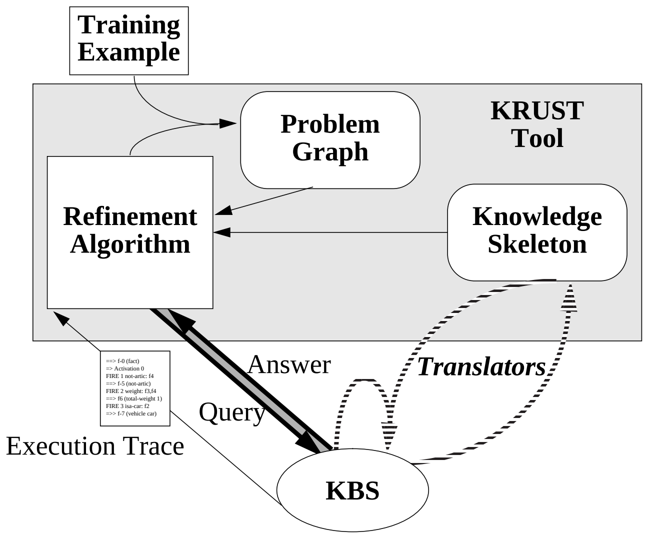
The core refinement procedures adopted by all KRUSTTools were shown in figure 1. Figure 3 places the refinement procedures in context, and shows how a KRUSTTool interacts with a KBS to create the knowledge skeleton and the problem graph. These provide the information needed to carry out refinement. The KRUSTTool performs the following steps.

1. The tool translates the KBS's rules into the knowledge skeleton, representing the static knowledge in the KBS.

2. The tool is given a training example, for which the KBS gives an incorrect solution, together with information about how the KBS reaches its conclusions. This information may be provided either in the form of an execution trace, or via queries submitted to the KBS. The tool uses the information to build a problem graph: an internal structure representing the reasoning of the KBS for the particular training example.

3. The refinement algorithm analyses the problem graph and the knowledge skeleton to determine where changes may be made to correct the errors made by the KBS. In general, the correct fix can not be uniquely determined, so the tool generates a number of alternative refinements, which are then filtered, implemented and tested. Testing consists of translating the modified knowledge skeletons back into the language of the KBS, and then executing them on the training example and others.

One distinguishing feature of KRUSTTools is that the refinement algorithm and the KBS run as separate processes. In contrast, EITHER [2] is written in PROLOG and refines only PROLOG KBSs, using a single process for both tool and KBS. The KRUSTTools approach is necessary for a generic refinement tool, since a separate refinement tool and KBS is necessary to allow the refinement of a KBS written in any language.



```
==> f-0 (fact)
=> Activation 0
FIRE 1 not-artic: f4
==> f-5 (not-artic)
FIRE 2 weight: f3,f4
==> f6 (total-weight 1)
FIRE 3 isa-car: f2
=>> f-7 (vehicle car)
```

Figure 3: Krust and KBS Processes

2 The Knowledge Skeleton Representation Language

The purpose of this language is to represent the rules in any KBS to which a KrustTool is to be applied. This seemingly over-ambitious task is made feasible because, despite the variety of different syntax and functionality apparently available in current KBS shells, there are in fact only a limited number of roles that a rule element can play [8, 9]. Furthermore, we are interested only in the ability to reason about which rules fire and the capability of the KrustTool to repair faults. Therefore, the knowledge representation language needs to represent faithfully only those parts of the knowledge that should be reasoned about and can be repaired. In contrast, for example, knowledge that contains external function calls does not need to be transformed into any special internal format since it will remain unchanged and so can be copied into any refined KBSs.

2.1 Basic Rule Elements

Each rule condition and conclusion is said to be a *rule element*. Three basic classes of rule element have been identified, corresponding to the fundamental roles they play in rules.

Tests can succeed or fail; e.g., retrievals from working memory, or comparisons such as `?amp-price` $\leq$ `?amp-budget` where `?var` is a variable name.

Expressions are rule elements that return a value, and always succeed; e.g., arithmetical calculations or function calls.

Assignments assign a value to a variable, and again always succeed.

These three basic classes form the first level of the hierarchy of rule elements shown in Figure 4.

It is clear that by defining an internal representation for each rule element type, we can create an internal data structure representing the static rules in the KBS. However, in addition to simply representing the rules, the knowledge skeleton must also allow a KrustTool to reason about the knowledge in order to refine it. The most direct way of changing the behaviour of any rule element is to change the rule element itself by applying an appropriate refinement operator. We therefore wish to establish a rule element hierarchy so that each leaf node is associated with a set of refinement operators that apply to rule elements of this type.

2.2 A Usable Knowledge Hierarchy

The initial partition above is too coarse for defining refinement operators. Therefore, we continue to partition these roles until each partition contains a class of rule element with a well-defined set of associated refinement operators. Each of the new nodes in the extended hierarchy of Figure 4 is now described.

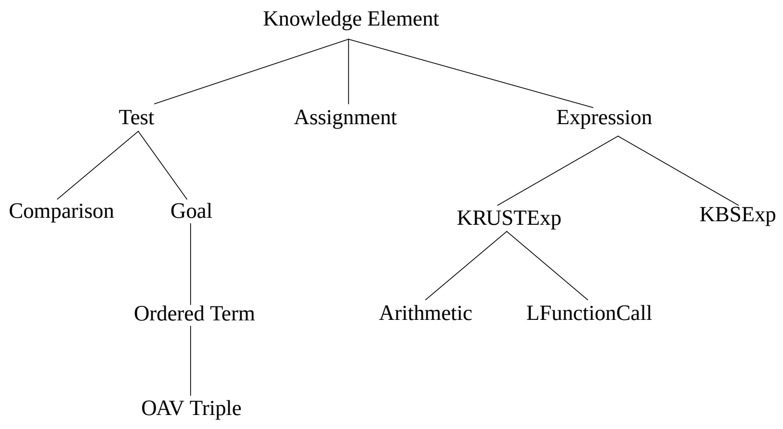


Figure 4: Hierarchy of Knowledge Elements

2.2.1 Tests

There are two sub-classes of test depending on whether the truth of the test comes from information local to the rule or knowledge deduced by other rules.

Comparisons are equations or inequalities. Their truth depends only on local information: the relational operator and the values being compared. Suitable refinement operators change the operator (`adjust-operator`) or the values (`adjust-value`).

Goals are rule elements that use the KBS's working memory; a conclusion that adds a fact to working memory, and a condition that uses that fact, are both classified as Goals[2]. We now identify two common sub-classes, but further sub-classes are added in Section 4.

Ordered Terms consist of a keyword followed by arguments; e.g., the PROLOG literal `colour(sky, blue, light)`.

OAV Triples are a sub-class of Ordered Term where the keyword is the attribute and the object and value form the remaining two arguments; e.g., `colour(sky, blue)`.

Goals offer another way to modify the behaviour of a rule's condition: by changing rules whose conclusions unify with the condition. Therefore, the knowledge skeleton must allow the KRUSTTool to determine when two Goals chain; i.e. a condition in one rule matches a conclusion of another rule. The `goals-match` function determines that two Ordered Terms (or OAV Triples) chain if and only if they have the same keyword and arity, and the corresponding arguments unify. As we propose further sub-classes of Goal we shall describe relevant refinement operators and modify the `goals-match` function.

2.2.2 Expressions

An expression is a piece of procedural knowledge which calculates and returns a value. There are two sub-classes, reflecting whether the calculation can be performed within the KRUSTTool or must be passed to the KBS for external execution.

KRUSTExps are evaluated within the KRUSTTool and are further divided.

Arithmetic expressions use the four standard arithmetical operators $+, -, \times, /$.

LFunctionCall expressions are Lisp functions and are executable in the KRUSTTool since it is implemented in Lisp. They represent KBS rule elements which are either written in Lisp or can be translated into Lisp. Alternatively, it would be possible to pass all non-arithmetical expressions to the KBS for evaluation, with the consequence however, that these expressions could not be refined.

KBSExps include all those expressions which cannot be evaluated within the KRUSTTool and so are passed to the original KBS for evaluation. KBSExps deal with situations where a KBS shell allows calls to procedural code, such as C functions, in rule elements. KBSExps cannot be refined.

[2]Describing Goals in terms of working memory applies particularly to forward-chaining rules. However, the distinguishing property of Goals is their ability to chain, so that we can identify and reason about Goals equally well for either backward or forward-chaining rules.

2.2.3 Assignments

Many rule elements, including those described so far, consist of a single element from the hierarchy, but a rule element can consist of an arbitrarily deep recursive structure. This is particularly relevant for Assignments and Arithmetic Expressions; e.g.,
`?amp-budget = (?budget - ?cd-price) * 0.6`
is an Assignment whose right-hand side is the Arithmetic Expression
`(?budget - ?cd-price) * 0.6`
These recursive style rule elements have also proved useful for more complex knowledge formats found in some KBS languages. Further examples appear in the following two sections.

We therefore now introduce the term *knowledge element* for the classes identified in the knowledge hierarchy, and reserve the term *rule element* for complete conditions or conclusions. Thus, rule elements are made up of one or more knowledge elements. We are therefore building a representation language for knowledge elements which can then be used to construct rule elements and hence rules.

3 Applying the Knowledge Skeleton

The hierarchy we have described (Figure 4) is fairly basic and was based on our experience with PROLOG KBSs and some simple KBSs written in CLIPS [10]. Figure 5 shows a rule broken down into the knowledge elements we have met in the previous section. We now investigate the hierarchy's expressiveness when applied to more advanced KBS shells. For this investigation we consider the knowledge structures available in three commercial KBS shells: CLIPS[3], POWERMODEL[4] and PFES[5] [11]. Both PFES and CLIPS use exclusively forward-chaining rules; POWERMODEL permits the use of both forward and backward-chaining rules. Many features of these shells corresponded to knowledge element types already present in the hierarchy, and this section explores the features to which the existing hierarchy was applicable.

3.1 CLIPS Patterns

CLIPS patterns correspond to Ordered Terms and provide rule chaining; e.g.,

```
(preferences amplifier denon-amp-40 cd marantz-cd-75)
(preferences amplifier ?amplifier cd marantz-cd-75)
```

Rules containing these elements as conclusion and condition respectively will chain. However, CLIPS has a more general wildcard than `?var`; `$?` matches 0 or more arguments. Such wild-cards require an appropriate extension to the `goals-match` function.

[3]CLIPS is an expert system shell widely used in both academia and industry.
[4]POWERMODEL is developed by IntelliCorp Ltd and is the successor to KAPPA.
[5]PFES (Product Formulation Expert System) is a development environment for KBSs that solve design and formulation problems.

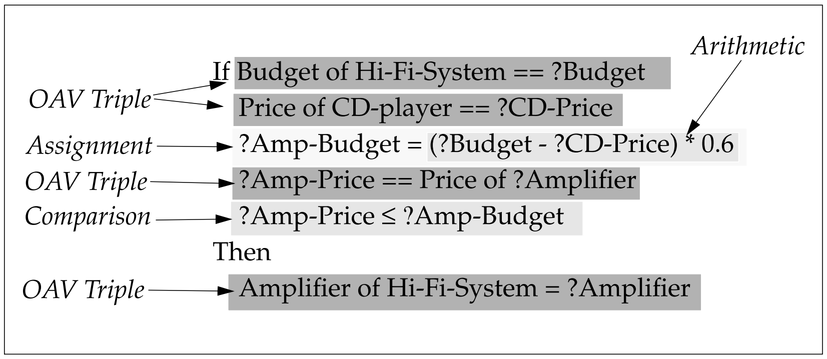

Arithmetic
If
Budget of Hi-Fi-System == ?Budget
OAV Triple
Price of CD-player == ?CD-Price
Assignment
?Amp-Budget = (?Budget - ?CD-Price) * 0.6
OAV Triple
?Amp-Price == Price of ?Amplifier
Comparison
?Amp-Price ≤ ?Amp-Budget
Then
OAV Triple
Amplifier of Hi-Fi-System = ?Amplifier

Figure 5: Sample Knowledge Elements in a Rule

3.2 PFES Agendas

There is a group of rule elements that appear at first unique to PFES, and therefore potentially difficult to represent within a common knowledge hierarchy. These *agendas* are untyped lists, where items can be read/written at the top or bottom, or directly below another given item. Agendas pass data between routines that generate values and those that subsequently test or filter them.

PFES agendas can also be viewed as a mechanism for storing attribute-value data; this better indicates how they may be represented within the existing hierarchy. Not all agendas have the same semantics, but the number of different possibilities actually employed within PFES applications is fairly limited. Two examples from the HI-FI program, whose purpose is to select the components of a hi-fi system, appear in Figure 6 where

speaker-agenda is a list of speakers; they are on it if they have the correct impedance

cd-price-agenda is a list of CD players, but now each item is associated with its price.

Each example shows the contents of an agenda at some point during the running of HI-FI, together with the PFES rule elements that write to and read from the agenda, and the KRUSTTool representation of these elements as Ordered Terms. Note that a conclusion that writes to the agenda, and the corresponding condition that reads from it, have the same KRUSTTool representation, though the two appear different in PFES.

Another feature of this representation is the fact that, while `?price`'s role as an attribute of `?cd-player` is implicit in the PFES statements, it is made explicit in the KRUSTTool representation, where both the CD player and its price are arguments. One consequence is that PFES commands of the type `add ?item to-bottom-of ?agenda` have different KRUSTTool representations, depending on whether `?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 Hi-fi.

Snapshot of speaker-agenda

```
ar-18bx dean-alto-11 castle-pembroke
```

Pfes **Read/Write Operations on speaker-agenda**

Conclusion:	`add ?speaker to-bottom-of speaker-agenda`
Condition:	`?speaker is-on speaker-agenda`

Krust**Tool representation of each operation**

```
on-agenda(speaker-agenda, ?speaker)
```

Snapshot of cd-price-agenda

```
philips-cd650 429 sony-cdp103 399 marantz-cd 429
```

Pfes **Read/Write Operations on cd-price-agenda**

Conclusions:	1) `add ?cd-player to-bottom-of cd-price-agenda`
	2) `add ?price to-bottom-of cd-price-agenda`
Conditions:	1) `?cd-player is-on cd-price-agenda`
	2) `?price is-the-item-after ?cd-player on cd-price-agenda`

Krust**Tool representation of each operation**

1) `on-agenda(cd-price-agenda, ?cd-player)`

2) `agenda-unlabelled-attribute(cd-price-agenda, ?cd-player, ?price)`

Figure 6: Agendas and their Pfes Operations

3.3 Compound Rule Elements

A powerful way to increase the expressiveness of the existing knowledge elements is to build compound rule elements from several knowledge elements. We met this idea already in Section 2.2.3 where we embedded an Arithmetic Expression as the body of an Assignment.

Clips, PowerModel, and many expert system shells allow conditions whose effect is to access a value and then test it; e.g., selecting CD players whose price is less than £200:

Clips	`(cd-player ?name ?price &: (< ?price 200))`
PowerModel	`?cd-player.price < 200;`

We have chosen to represent these as compound rule elements; here, a Comparison embedded in an Ordered Term: `(cd-player ?name (?price < 200))`.

This has consequences for `goals-match` and the refinement operators, which need to decompose the rule elements to which they are applied. In this example, the refinement operators for Ordered Terms are applicable to the term, but will not affect the Comparison element, while the refinement operators for Comparison are applicable to the Comparison nested within the term.

4 Extensions to the Hierarchy

We encountered several rule element types which could not be represented by the elements of the existing hierarchy shown in Figure 4. However, these new knowledge elements were found to be more specialised versions of existing knowledge elements and so could be added to the hierarchy without any revision to the basic structure. Figure 7 shows how the hierarchy has been expanded by the addition of the new Goal sub-class AV Tuple, described below.

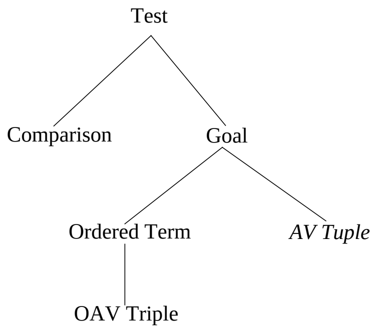


Figure 7: Adding a New Term to the Knowledge Element Hierarchy

4.1 Attribute Value Tuples

Many shells have a frame-based knowledge representation. Frames are similar to Ordered Terms, but the significance of each argument is determined by a preceding keyword rather than by the argument's position in a list; e.g., a CLIPS `amplifier` frame can be defined as:

```
(deftemplate amplifier
  (slot name)
  (slot price)
  (slot power)
  (slot impedance)
```

and a rule condition which tests the properties of `amplifier` looks like

```
(amplifier (name ?name) (power ?power) (price ?price))
```

The inherently unordered nature of this condition precludes its representation as an Ordered Term, so a new sub-class of Goal called an AV (Attribute-Value) Tuple was introduced. The AV Tuple consists of a keyword followed by a series of attribute-value pairs, so that the condition above is represented by the AV Tuple

```
(amplifier name ?name power ?power price ?price)
```

and matches conclusions such as

```
(amplifier name marantz-pm26 price 125 power 30 impedance 8)
```

The `goals-match` function is defined to use keywords rather than order for matching, and refinement operators are adapted so that they do not alter keywords.

4.2 Future Knowledge Elements

PowerModel permits the use of a variety of iterative operators within its rule conditions. A typical example is the loop:

```
for find ?x = instanceof Tuner;
    do ?x.presets = 7;
```

which retrieves each instance of `Tuner`, and sets its `presets` slot to 7. Currently, knowledge like this is simply copied verbatim in the knowledge skeleton, ignored during refinement generation, and re-created in its original form in the refined KBSs. However, this approach may be unnecessarily restrictive, given that within the procedural "wrapping" there are statements which *can* be refined. This is an area requiring further work, but the `find` loop above suggests the following. The statement contains an Assignment which can have an Assignment refinement operator applied. For example, suppose for the training example `?x` is bound to `quad-fm4` and its presets should be `9`, then the refined knowledge might look as follows:

```
for find ?x = instanceof Tuner;
    do if ?x = quad-fm4
       then ?x.presets = 9
       else ?x.presets = 7;
```

5 Comparison with Other Work

Johnson and Carlis [9] have also classified the rule elements in expert systems shells, but have taken a more syntax based approach. Their work confirms our view that it is possible to build a common representation for expert system shells while avoiding the need to introduce particular shell-specific items, and hence the feasibility of our generic approach to refinement. We now consider the restrictions imposed on KBSs by other refinement tools.

Either [2] and Forte [3] rely on Prolog's Horn Clauses for their reasoning and ignore the control imposed by Prolog's depth first search when multiple solutions are available. Each condition in a Prolog rule is a Prolog literal, and corresponds to an Ordered Term in the KrustTool hierarchy; the literal's predicate name is the Ordered Term's keyword. Prolog's depth first search of clauses easily provides information for our problem graph. Therefore Prolog KBSs have KrustTool representations.

Neither [12] extends Either's refinement process by having specialised refinement operators for m-of-n rules. An *m-of-n* condition contains a set of n conditions, and is defined to be true if and only if at least m of the n conditions are true. Similarly, Seek [5] refines rules in a specialised form where normal conditions are supplemented with m-of-n type conditions, but now the n conditions are symptoms associated with a diagnosis rather than explicitly listed in the condition. An m-of-n condition is a new Goal sub-class requiring a specialised `goals-match` function and specific refinement operators.

Clips-R [4] is similarly restrictive, since it refines only Clips KBSs. It uses example traces to build a data structure which groups together those examples that share an initial sequence of rule firings. This data structure guides Clips-R towards the most common errors. Since the Clips-R data structure represents the execution on training examples, it is similar in purpose to our problem graph.

Odysseus [13] illustrates a different approach to the use of control information from that of the other programs surveyed. Meta-rules contain the control knowledge, and failure to solve problems is attributed to missing domain knowledge which should have been available to be used by the control knowledge. By representing the control knowledge explicitly, Odysseus is able to guide the refinement process.

6 Conclusions

The knowledge element hierarchy we have developed has been shown to provide a powerful representation mechanism for rule-based KBSs. It has evolved in a disciplined way from experience with several basic KBSs. Many of the new constructions found in more sophisticated KBSs have been directly equivalent to existing knowledge elements; e.g. Pfes agendas. However, the hierarchy is also extensible in a natural way, by incorporating novel rule elements in two ways: new knowledge elements have extended the hierarchy without destroying its basic structure; or a recursive structure of existing knowledge elements represents the new rule element. It has thus been shown to be able to accommodate novel rule elements from a variety of shells.

In this paper we have concentrated on the feasibility of a knowledge element hierarchy as the basis of a representation language for knowledge skeletons. Knowledge skeletons contain the essential knowledge content of a KBS, and the hierarchy additionally identifies suitable refinement operators for the knowledge elements. Therefore, it is possible to have a common core of routines for blame allocation, refinement generation and filtering that explore the problem graph and manipulate the knowledge skeleton. We have not been concerned here with the efficacy of refinement, but other papers have shown KrustTools being successfully applied to a range of KBSs. The Prolog-based student loan rules [14] have been translated into Clips and PowerModel, and KrustTools have been applied to fix artificially introduced faults in all 3 versions [15, 16]. A KrustTool has successfully been applied to the debugging and maintenance of the Pfes-based tablet formulation system Tfs developed by Zeneca Pharmaceuticals [17, 18].

The KrustWorks project is applying these ideas to provide a framework of re-usable refinement components from which to assemble a KrustTool for a particular KBS. Figure 8 illustrates the process. The knowledge engineer provides the rulebase and interpreter for a KBS, together with a grammar for

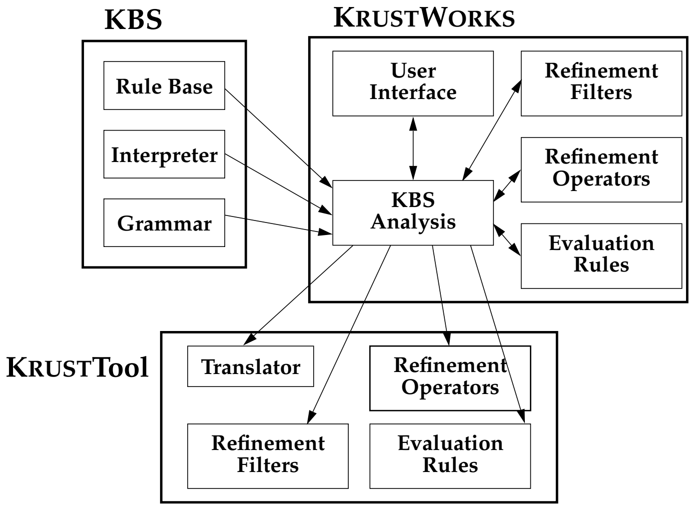


Figure 8: Creating a KrustTool from the KrustWorks Framework

parsing the rules. KrustWorks performs an analysis of the KBS, determining properties such as the kinds of rule elements present, and the direction of rule chaining. Guided by the knowledge engineer, KrustWorks generates and customises modules to perform translation and the standard refinement tasks. The resulting specific KrustTool, shown at the bottom of the diagram, is customised to the needs of the particular application and does not contain unnecessary functionality.

Acknowledgements

This work is supported by EPSRC grant GR/L38387 awarded to Susan Craw.

We also thank Intellicorp Ltd. for donating and supporting our PowerModel licences.

References

[1] G. Schreiber, Bob Wielinga, and J. Breuker, editors. KADS: A Principled Approach to Knowledge Based Systems Development. Academic Press, 1993.

[2] Dick Ourston and Raymond Mooney. Theory refinement combining analytical and empirical methods. Artificial Intelligence, 66:273–309, 1994.

[3] Bradley L. Richards and Raymond J. Mooney. Refinement of first-order horn-clause domain theories. Machine Learning, 19(2):95–131, 1995.

[4] Patrick M. Murphy and Michael J. Pazzani. Revision of production system rule-bases. In W. W. Cohen and H. Hirsh, editors, Proceedings of the Eleventh International Conference on Machine Learning, pages 199–207, New Brunswick, NJ, 1994. Morgan Kaufmann.

[5] Allen Ginsberg. Automatic Refinement of Expert System Knowledge Bases. Research Notes in Artificial Intelligence. Pitman, London, 1988.

[6] Susan Craw. Refinement complements verification and validation. International Journal of Human-Computer Studies, 44(2):245–256, 1996.

[7] Susan Craw and D. Sleeman. Automating the refinement of knowledge-based systems. In L. C. Aiello, editor, Proceedings of the ECAI90 Conference, pages 167–172, Stockholm, Sweden, 1990. Pitman.

[8] Gareth Palmer and Susan Craw. An extensible knowledge refinement tool. Technical Report 96/2, SCMS, Robert Gordon University, 1996.

[9] Verlyn M. Johnson and John V. Carlis. Building a composite syntax for expert system shells. IEEE Expert, 12(6):60–66, 1997.

[10] Joseph C. Giarratano. Expert Systems : Principles and Programming. International Thomson, 3rd edition, 1998.

[11] Alvey. The PFES report, volume three: The formulations kernel. Logica UK Ltd, 1987.

[12] Paul T. Baffes and Raymond J. Mooney. Symbolic revision of theories with M-of-N rules. In Ruzena Bajcsy, editor, Proceedings of the Thirteenth IJCAI Conference, pages 1135–1140, Chambery, FRANCE, 1993.

[13] David C. Wilkins. Knowledge base refinement as improving an incorrect and incomplete domain theory. In Y. Kodratoff and R. S. Michalski, editors, Machine Learning: An Artificial Intelligence Approach Volume III, pages 493–513. Morgan Kaufmann, San Mateo, CA, 1990.

[14] Michael J. Pazzani. Student loan relational domain. In UCI Repository of Machine Learning Databases [19], 1993.

[15] Gareth Palmer. Applying KRUST to a new KBS tool: experience with Kappa. Technical Report 95/9, SCMS, Robert Gordon University, October 1995.

[16] Gareth J. Palmer and Susan Craw. The role of test cases in automated knowledge refinement. In Proceedings of the 16th Annual Technical Conference of the British Computer Society Specialist Group on Expert Systems, pages 75–90, Cambridge, UK, 1996. SGES Publications.

[17] Susan Craw, Robin Boswell, and Ray Rowe. 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), pages 446–453, Newport Beach, CA, 1997. IEEE Press.

[18] Robin Boswell. Knowledge Refinement for a Formulation System. PhD thesis, School of Computer and Mathematical Sciences, The Robert Gordon University, 1998.